NYSE +0.31% · NASDAQ +0.47% · LSE -0.12% · Eurex +0.18% · JPX +0.22% · HKEX -0.08% · CME +0.55% · SGX +0.11%



EQUITY TRADING INFRASTRUCTURE

ARCHITECTURE AND DEVELOPER GUIDE

A Guide to the architecture and development of EduMatcher

MARKET DEPTH · AAPL +1.24 +0.83%

BID QTY		PRICE	ASK QTY
8,340		150.35	3,920
7,210		150.34	5,100
5,820		150.33	6,400
12,100	LAST	150.32	7,840
4,420		150.31	10,220
3,200		150.30	12,800

EduMatcher Architecture and Developer Guide

For EduMatcher Version: v0.77.0

Copyright © Johan Persson

Generated on November 8, 2026

Prepared with LaTeX from Markdown sources for duplex printing. Sources for documentation available at https://github.com/johan162/EduMatcher.

Architecture and Developer Guide

`edumatcher`, Version: v0.77.0

System architecture, development practice, and release engineering

Johan Persson <johan162@gmail.com>

Generated on November 8, 2026

Chapters

1 How to Use This Book 1

2 Architecture 3

3 Guided Tour: Architecture, Data Structures and Code 37

4 SQLite Database Formats 85

5 SQLite Schema Reference 87

6 The Order Book: Deep Dive 121

7 Order Flow Through the Engine: A NEW LIMIT Order, Step by Step 165

8 Development Practice 181

9 The Development Loop 193

10 Deterministic Verification of the Matching Engine 201

11 Message Generator (`pm-msgen`) 213

12 Container and Network Setup 235

13 Exchange Experiments Roadmap 255

14 The Release Process 277

Contents

1 How to Use This Book **1**
- 1.1 How the book is organized . . . 1
- 1.2 Where to start . . . 1

2 Architecture **3**
- 2.1 Why This Architecture? . . . 3
  - 2.1.1 Three ways to build a trading system . . . 3
  - 2.1.2 Core messaging concepts . . . 4
- 2.2 Overview . . . 4
- 2.3 ZMQ Topology . . . 4
- 2.4 Message Topics . . . 4
  - 2.4.1 Causal envelope . . . 5
  - 2.4.2 Commands — GW / operator → Engine (PUSH :5555) . . . 5
  - 2.4.3 Private replies — Engine → GW (PUB :5556, personalised prefix) . . . 6
  - 2.4.4 Broadcasts — Engine → all subscribers (PUB :5556) . . . 7
  - 2.4.5 Drop-copy feed — Engine → compliance consumers (PUB :5557) . . . 7
  - 2.4.6 Index bus — pm-index clients (PUB :5558 / PULL :5559) . . . 7
  - 2.4.7 Market-model bus — pm-market-sim clients (PUB :5553 / PULL :5554) . . . 8
- 2.5 Process Roles . . . 8
- 2.6 Data Flows . . . 10
- 2.7 Key Design Decisions . . . 11
  - 2.7.1 The Engine as the Sole Binder . . . 11
  - 2.7.2 Two Tiers of Gateways . . . 11
  - 2.7.3 Bus Segmentation by Function . . . 11
  - 2.7.4 Single-Threaded Engine as a Correctness Guarantee . . . 12
  - 2.7.5 Deployment Topology & Trust Boundaries . . . 12
- 2.8 Order Book Data Structures . . . 13
  - 2.8.1 Visual: order depth at each price level . . . 13
  - 2.8.2 Internal representation . . . 13
- 2.9 Thread Model . . . 14
- 2.10 Core Matching Algorithm — In Depth . . . 14
  - 2.10.1 Order Book Organization . . . 14
  - 2.10.2 Heap Entry and Lazy Deletion . . . 14
  - 2.10.3 Matching Algorithm: The Sweep . . . 15
  - 2.10.4 Apply Fill . . . 16
  - 2.10.5 Stop Order Trigger Mechanism . . . 16
  - 2.10.6 Resting an Order . . . 17
  - 2.10.7 Cancellation . . . 17
  - 2.10.8 Time Complexity Summary . . . 17
  - 2.10.9 Combo Orders — Data Structures and Tracking . . . 17
  - 2.10.10 Session State Machine . . . 21
  - 2.10.11 Auction Uncross Algorithm — Equilibrium Price . . . 22

2.10.12 Gateway Authentication . . . 24
2.10.13 Engine Event Loop . . . 24
2.11 Persistence & Crash Recovery . . . 25
2.12 Failure Modes & Fault Tolerance . . . 26
2.12.1 Back-pressure: PUSH blocks, PUB drops . . . 27
2.12.2 Other failure modes . . . 27
2.13 Performance Optimizations . . . 28
2.13.1 How to read the numbers . . . 28
2.13.2 `__slots__` on hot-path classes . . . 29
2.13.3 Fast enum lookup dictionaries . . . 29
2.13.4 Single timestamp call per order (cached nanosecond clock) . . . 30
2.13.5 Monotonic integer trade IDs (replacing `uuid4()`) . . . 30
2.13.6 `orjson` instead of stdlib `json` . . . 30
2.13.7 Eliminate redundant serialization (`to_dict()` removal) . . . 31
2.13.8 Inlined message construction . . . 31
2.13.9 Pre-cached topic bytes . . . 32
2.13.10 Local variable caching in tight loops . . . 32
2.13.11 `__new__` for fast deserialization . . . 32
2.13.12 Summary . . . 33
2.13.13 Why ARM and Intel differ so much . . . 33
2.14 Configuration Choices That Impact Performance . . . 34
2.15 Tick And Time Representation . . . 35

3 Guided Tour: Architecture, Data Structures and Code 37
3.1 Quick-Reference Glossary . . . 37
3.2 Table of Contents . . . 38
3.3 How to Use This Document . . . 38
3.4 1. What Is an Exchange? The One-Sentence Version . . . 39
3.5 2. The Limit Order: The Atom of Trading . . . 39
3.5.1 Why `__slots__`? . . . 39
3.6 3. The Order Book: A Sorted List of Promises . . . 41
3.7 4. The First Match: How Two Orders Become a Trade . . . 44
3.7.1 What Happens to `trades` and `events` After `process()` Returns? . . . 46
3.7.2 The Partial Fill Case . . . 46
3.8 5. Data Structures: Why Heaps? . . . 47
3.8.1 The Problem . . . 47
3.8.2 The Heap . . . 47
3.8.3 The `_HeapEntry` Wrapper . . . 47
3.8.4 Lazy Deletion . . . 47
3.8.5 The Dual-Index Pattern . . . 48
3.8.6 The Price-Level Quantity Index . . . 49
3.8.7 Summary: Why This Combination? . . . 49
3.9 6. The Order Model: Every Field Explained . . . 49
3.9.1 Time-In-Force (TIF): How Long Does the Order Live? . . . 51
3.9.2 The Enum Design Pattern . . . 51
3.9.3 The `from_dict` Bypass . . . 51
3.10 7. Beyond Limit Orders: The Full Order Type Taxonomy . . . 52
3.10.1 Market Order . . . 52
3.10.2 IOC (Immediate-Or-Cancel) . . . 52
3.10.3 FOK (Fill-Or-Kill) . . . 52
3.10.4 Stop Orders . . . 52
3.10.5 STOP vs STOP_LIMIT: A Critical Difference . . . 53
3.10.6 Trailing Stop Orders . . . 53
3.10.7 Iceberg Orders . . . 54
3.11 8. The Trading Day: Auctions and Continuous Matching . . . 55
3.11.1 The Scheduler: A Separate Process . . . 55
3.11.2 No-Matching Mode: Collecting Auction Interest . . . 56

3.11.3 The Auction Uncross . . . 56
3.12 9. Complex Orders: Combos and OCO . . . 57
3.12.1 OCO (One-Cancels-Other) . . . 57
3.12.2 Combo Orders (Multi-Leg) . . . 59
3.13 10. Market-Maker Quotes: Two Orders, One Obligation . . . 60
3.13.1 The Two Orders . . . 60
3.13.2 Index 1 — `QuoteIndex`: Which Two Orders Make Up *This* Quote? . . . 61
3.13.3 Index 2 — `OrderBook._orders_by_gateway`: Which Orders Are *Still Resting*, Per Gateway? . . . 61
3.13.4 A Second Consumer: Mass-Cancel on Disconnect . . . 62
3.13.5 Why Two Indexes, Not One . . . 62
3.13.6 The Fallback: Cancelling a Leftover Leg `QuoteIndex` No Longer Knows About . . . 63
3.13.7 Walkthrough: A Fill, a Stray Leg, and a Reissue . . . 64
3.13.8 Restart: Both Indexes Rebuild From the Same Source . . . 64
3.14 11. The Message Bus: Why ZeroMQ? . . . 64
3.14.1 The Problem with Direct Function Calls . . . 64
3.14.2 The Publish-Subscribe Pattern . . . 65
3.14.3 Topic-Based Filtering . . . 65
3.14.4 PUSH/PULL for Order Submission . . . 66
3.14.5 Message Format . . . 66
3.14.6 Why ZeroMQ Specifically? . . . 67
3.15 12. The Full Process Architecture . . . 68
3.15.1 The Engine: Single-Threaded by Design . . . 69
3.15.2 The Gateway: User Interface . . . 69
3.15.3 Clearing: Tracking Positions and P&L . . . 70
3.15.4 Stats: OHLCV and Market Data . . . 71
3.15.5 Audit: The Complete Event Log . . . 71
3.16 13. Optimisations: Speed, Memory, and Latency . . . 71
3.16.1 The Latency Budget . . . 71
3.16.2 Architecture: Single Timestamp Per Order . . . 72
3.16.3 `__slots__` on All Hot-Path Objects . . . 73
3.16.4 Pre-Built Enum Lookup Dicts . . . 73
3.16.5 `from_dict` via `__new__` Bypass . . . 73
3.16.6 Pre-Cached ZMQ Topic Bytes . . . 73
3.16.7 Inlined Message Dicts on Hot Paths . . . 74
3.16.8 Throttled Book Snapshot Publishing . . . 74
3.16.9 Trade ID: Monotonic Counter Instead of UUID . . . 75
3.16.10 Local Variable Caching in the Inner Loop . . . 75
3.16.11 `orjson` for Serialization . . . 75
3.16.12 FOK Pre-Check . . . 75
3.17 14. Persistence: Surviving a Restart . . . 76
3.17.1 What Persistence Does NOT Cover . . . 77
3.18 15. Clearing and Settlement: Following the Money . . . 77
3.19 16. What Is Missing: The Gap to Production . . . 78
3.19.1 Guaranteed Delivery and Replay on the Core Message Bus . . . 78
3.19.2 Authentication and Authorisation . . . 79
3.19.3 TLS Encryption . . . 79
3.19.4 Primary and Secondary (Failover) Sites . . . 80
3.19.5 Order Replay and Recovery . . . 80
3.19.6 Rate Limiting and Risk Checks . . . 80
3.19.7 High-Precision Timestamps and Clock Synchronisation . . . 81
3.19.8 Price Precision . . . 81
3.19.9 Network Level: FIX Protocol . . . 81
3.19.10 Regulatory Requirements . . . 81
3.19.11 Summary Table . . . 82
3.20 Conclusion . . . 82

4 SQLite Database Formats 85

4.1 Storage Behavior . . . 85
4.2 SQLite Files on Disk . . . 85
4.3 Keeping the Schema Reference Current . . . 86

5 SQLite Schema Reference 87
5.1 stats.db . . . 87
5.1.1 Fields . . . 87
5.1.2 DDL . . . 91
5.2 clearing.db . . . 95
5.2.1 Fields . . . 95
5.2.2 DDL . . . 98
5.3 eod.db . . . 102
5.3.1 Fields . . . 102
5.3.2 DDL . . . 102
5.4 log.db . . . 103
5.4.1 Fields . . . 103
5.4.2 DDL . . . 104
5.5 metrics.db . . . 105
5.5.1 Fields . . . 105
5.5.2 DDL . . . 107
5.6 market_sim_news.db . . . 108
5.6.1 Fields . . . 108
5.6.2 DDL . . . 109
5.7 companies.db . . . 109
5.7.1 Fields . . . 109
5.7.2 DDL . . . 112
5.8 log-ui-acks.db . . . 112
5.8.1 Fields . . . 112
5.8.2 DDL . . . 113
5.9 audit_index.db . . . 113
5.9.1 Fields . . . 114
5.9.2 DDL . . . 114
5.10 audit_replay.db . . . 115
5.10.1 Fields . . . 115
5.10.2 DDL . . . 117

6 The Order Book: Deep Dive 121
6.1 Where the File Sits . . . 121
6.2 What Is an Order Book? . . . 122
6.3 Background: The Heap Data Structure . . . 123
6.4 The Priority Key: (±price, arrival_seq) . . . 123
6.4.1 arrival_seq is the clock, not ts_ns . . . 124
6.5 _HeapEntry: Why the Heap Does Not Hold Orders . . . 124
6.6 The OrderBook Object at a Glance . . . 126
6.7 Why Five Side Indexes? . . . 126
6.7.1 _order_index — the id lookup . . . 127
6.7.2 _entry_index — the handle for lazy deletion . . . 127
6.7.3 _orders_by_gateway — "everything gateway X has here" . . . 127
6.7.4 _bid_qty / _ask_qty — display depth per price level . . . 128
6.7.5 Which order lives where . . . 128
6.8 The Rest of the State . . . 129
6.8.1 Stop structures . . . 129
6.8.2 Market statistics . . . 129
6.8.3 Engine hooks . . . 130
6.9 Lazy Deletion and Compaction . . . 130
6.9.1 Garbage that never reaches the top . . . 130
6.9.2 _purge_from_indexes — the single exit door . . . 131

6.9.3 Seen in a real run . . . 131
6.10 Synchronisation Rules . . . 131
6.11 The Order Lifecycle . . . 133
6.12 The Entry Point: `process()` . . . 133
6.12.1 Step by step through the code . . . 134
6.12.2 What comes back . . . 135
6.13 The Matchers . . . 136
6.13.1 LIMIT — `_match_limit` . . . 136
6.13.2 MARKET — `_match_market` . . . 136
6.13.3 IOC — `_match_ioc` . . . 136
6.13.4 FOK — `_match_fok` . . . 136
6.13.5 ICEBERG — `_match_iceberg` . . . 137
6.13.6 STOP, STOP_LIMIT, TRAILING_STOP . . . 137
6.14 The Matching Loop: `_sweep()` . . . 137
6.15 Where Every Trade Is Born: `_apply_fill` . . . 139
6.15.1 The aggressor side . . . 140
6.15.2 The passive side . . . 140
6.15.3 The callbacks . . . 140
6.15.4 The Three Last Prices . . . 141
6.16 Worked Example: One LIMIT Order, Step by Step . . . 141
6.17 Resting an Order: `_rest()` . . . 142
6.18 Cancelling: `cancel_order()` . . . 142
6.19 Amending: `amend_order()` . . . 143
6.20 Icebergs . . . 144
6.20.1 Worked example (real output) . . . 144
6.21 Self-Match Prevention (SMP) . . . 145
6.22 Stop Orders . . . 146
6.22.1 Arming: `_add_stop` . . . 146
6.22.2 Triggering: `_check_stops` . . . 147
6.22.3 Worked example: a stop cascade (real output) . . . 147
6.23 Trailing Stops . . . 148
6.24 OCO Orders and Quote Legs . . . 148
6.25 No-Matching Mode and Auctions . . . 149
6.26 Self-Match Prevention in an Auction Uncross . . . 149
6.26.1 What the SMP walk is for . . . 149
6.26.2 The rules, exactly . . . 150
6.26.3 How the price is chosen with the walk . . . 150
6.26.4 Executing the uncross . . . 151
6.26.5 Why a resolve pass is needed: the pair the walk never meets . . . 151
6.26.6 A full walk-through (no price conflict) . . . 151
6.26.7 A harder walk-through: the curves overstate by a factor of ten . . . 152
6.26.8 Complications this adds . . . 153
6.26.9 Where this lives in the code . . . 153
6.27 Trace Records: What the Book Reports Besides Trades . . . 154
6.28 Read-Only Views . . . 154
6.29 Hot-Path Risk Checks Around the Book . . . 155
6.30 Learning by Debugging . . . 157
6.30.1 1. A book lab script . . . 157
6.30.2 2. Debugger configurations . . . 158
6.30.3 3. Where to put breakpoints . . . 159
6.30.4 4. Watch expressions that do not change the book . . . 159
6.30.5 5. Reading the debug log . . . 160
6.30.6 6. Experiments to try . . . 160
6.30.7 7. Gotchas while debugging . . . 160
6.31 Where to Look in the Code . . . 160
6.32 Data Structure Summary . . . 161
6.33 Complexity Summary . . . 162

7 Order Flow Through the Engine: A NEW LIMIT Order, Step by Step **165**
7.1 1. The moving parts . . . 165
7.2 2. The scenario . . . 167
7.3 3. From keystroke to the engine's doorstep . . . 167
7.4 4. The engine's front door . . . 168
7.5 5. The gauntlet: every check, in order . . . 169
7.6 6. The ACK — sent before the book ever sees the order . . . 171
7.7 7. Handing the order to the book . . . 173
7.7.1 7a. Outcome 1 — the order rests (our scenario) . . . 174
7.7.2 7b. Outcome 2 — a crossing order (worked example) . . . 174
7.8 8. Publishing what happened . . . 175
7.9 9. TIF and the lifetime of a resting order . . . 176
7.9.1 What happens at shutdown — and it is *not* removal . . . 177
7.9.2 What happens at the next startup — this is where staleness is decided . . . 177
7.9.3 Summary table . . . 178
7.10 10. Debugging cheat-sheet . . . 178
7.10.1 Following a chain . . . 179
7.11 11. The whole thing, end to end . . . 179

8 Development Practice **181**
8.1 What kind of project you are joining . . . 181
8.2 Development environment . . . 181
8.2.1 Recommended Python version . . . 181
8.2.2 Canonical setup . . . 182
8.2.3 Basic toolchain expectations . . . 182
8.3 Repository map . . . 182
8.3.1 Client libraries: `alf_client` and `calf_client` . . . 183
8.3.2 Good first files to read . . . 184
8.4 Development workflow expectations . . . 184
8.4.1 Default working style . . . 184
8.4.2 Code quality gates . . . 184
8.4.3 Standards enforced by the repo . . . 184
8.4.4 What to preserve . . . 185
8.5 Running a minimal system while developing . . . 185
8.5.1 Where your data directory actually is . . . 185
8.5.2 Minimal reference data . . . 186
8.5.3 Recommended startup order . . . 186
8.5.4 Inspecting a run after the fact . . . 187
8.5.5 Logging levels and pm-log-srv . . . 187
8.5.6 macOS convenience launcher . . . 188
8.5.7 Quick signs the system is healthy . . . 188
8.6 Verification and test strategy . . . 188
8.6.1 Normal regression tests . . . 188
8.6.2 Deterministic engine verification . . . 188
8.6.3 Performance tests . . . 189
8.6.4 When to run which check . . . 189
8.7 Helper scripts under `scripts/` and `tools/` . . . 189
8.7.1 Core scripts . . . 189
8.8 Documentation workflow . . . 190
8.8.1 Fast documentation loop . . . 190
8.8.2 One-shot validation . . . 190
8.8.3 What to update when behavior changes . . . 190
8.9 Mermaid diagrams in the docs build (cached Pandoc filter) . . . 191
8.9.1 Why a custom filter . . . 191
8.9.2 Why it isn't just a patch to the installed package . . . 191
8.9.3 How the cache works . . . 191

9 The Development Loop **193**
9.1 Summary . . . 193
9.2 Getting started: from a fresh clone . . . 193
9.3 Part 1 — Why `make up-all` does not pick up your change . . . 194
9.4 Part 2 — The hybrid loop, and why it needs no wiring . . . 195
9.4.1 `make dev-env` . . . 195
9.4.2 Ports at a glance . . . 195
9.5 Part 3 — Per-application recipes . . . 196
9.5.1 terminal-gui . . . 196
9.5.2 log-gui . . . 197
9.5.3 trader-gui . . . 197
9.5.4 config-gui . . . 197
9.5.5 dash-gui . . . 197
9.6 Part 4 — Choosing a loop . . . 197
9.6.1 Editing shared packages . . . 198
9.7 Part 5 — Things that will bite . . . 198
9.8 Reference . . . 199

10 Deterministic Verification of the Matching Engine **201**
10.1 Why verification is hard . . . 201
10.1.1 The matching engine is not a pure function . . . 201
10.1.2 Replay vs. re-simulation . . . 201
10.2 The verification flow . . . 202
10.3 Tool 1 — `gen_verification_set.py` . . . 202
10.3.1 What it builds . . . 202
10.3.2 Market-maker seed orders . . . 202
10.3.3 Random test orders: new, amend, or cancel . . . 203
10.3.4 The paper trader . . . 204
10.3.5 Determinism is guaranteed by the seed . . . 204
10.3.6 Typical output . . . 205
10.4 Tool 2 — `replay_to_engine.py` . . . 205
10.4.1 Prerequisites . . . 205
10.4.2 Protocol . . . 205
10.4.3 Snapshot collection . . . 206
10.4.4 Typical output . . . 206
10.5 Tool 3 — `compare_results.py` . . . 206
10.5.1 What is compared . . . 206
10.5.2 What is deliberately **not** compared . . . 206
10.5.3 Tolerance flag . . . 207
10.5.4 Reading the output . . . 207
10.6 Tool 4 — `verify_matching.sh` . . . 207
10.6.1 Command-line flags . . . 208
10.6.2 Clean-state guarantee . . . 208
10.7 The hardest problems: time, identity, and state . . . 208
10.7.1 Problem 1 — Order ids, now reproducible by design . . . 208
10.7.2 Problem 2 — TIF=DAY orders expire at session end . . . 208
10.7.3 Problem 3 — STOP prices depend on last_trade_price sequence . . . 209
10.7.4 Problem 4 — Gateway IDs differ between paper and engine . . . 209
10.7.5 Problem 5 — The drain window . . . 209
10.7.6 Problem 6 — Iceberg visible-qty refresh . . . 209
10.8 The verification config . . . 209
10.9 Running the full suite . . . 210
10.10 Summary . . . 211
10.11 What next? . . . 212

11 Message Generator (`pm-msgen`) **213**
11.1 The problem . . . 213

11.2 How it fits together . . . 214
11.3 Commands . . . 214
11.3.1 The normal edit loop . . . 214
11.4 Adding a family . . . 215
11.4.1 Two comparisons, not one . . . 215
11.5 Writing a specification . . . 215
11.5.1 Family-level keys . . . 215
11.5.2 A minimal family . . . 216
11.5.3 Message-level keys . . . 216
11.5.4 Field keys . . . 217
11.5.5 Presence: four regimes, stated explicitly . . . 218
11.5.6 Why `unit` is mandatory on numbers . . . 218
11.5.7 Records: `types:`, `nested` and `list` . . . 219
11.5.8 Topics with parameters . . . 220
11.5.9 Transports . . . 220
11.6 Binary messages . . . 220
11.6.1 Rules the loader enforces . . . 221
11.6.2 Using it . . . 222
11.7 Projections: one event, several shapes . . . 222
11.8 Generated C . . . 223
11.8.1 What a projection looks like in C . . . 224
11.8.2 Example 7 — reading a trade in C . . . 224
11.8.3 Building against the generated headers . . . 224
11.8.4 Error codes . . . 225
11.9 How the two bindings are kept honest . . . 225
11.10 Using a generated binding . . . 225
11.10.1 What you get . . . 225
11.10.2 Example 1 — publishing . . . 226
11.10.3 Example 2 — consuming . . . 227
11.10.4 Example 3 — validating without parsing . . . 227
11.10.5 Example 4 — subscribing without a topic literal . . . 227
11.10.6 Example 5 — runtime field metadata . . . 227
11.10.7 Example 6 — the hot path . . . 228
11.11 The cost of the hot path . . . 228
11.12 Adopting a binding in existing code . . . 229
11.12.1 A delegating shim . . . 229
11.12.2 The producer itself . . . 229
11.12.3 A recorder: topic constant, not the parser . . . 230
11.13 Coercion and validation are different jobs . . . 230
11.14 The guarantee: `pm-msgen check` . . . 231
11.14.1 Where it runs . . . 231
11.15 Why the generated file looks the way it does . . . 231
11.16 Migrating a family's topic literals . . . 232
11.17 Testing . . . 232
11.18 Scope . . . 233
11.19 See also . . . 233

12 Container and Network Setup **235**
12.1 Summary . . . 235
12.1.1 Compose file map . . . 236
12.2 Part 1 — How the Makefiles are structured . . . 237
12.2.1 What the app Makefiles give you . . . 238
12.2.2 Variable names that mean different things . . . 238
12.3 Part 2 — Networking: who is responsible for what . . . 238
12.3.1 Bind address versus connect address . . . 238
12.3.2 The four layers an address passes through . . . 240
12.3.3 Why the GUIs need no host address at all . . . 241

12.3.4 The two planes and their defaults . . . 241
12.3.5 How a service listener resolves its bind host . . . 242
12.3.6 Why the entrypoint no longer rewrites anything . . . 242
12.4 Part 3 — Build time versus run time . . . 242
12.4.1 Precedence when the same thing is settable in several places . . . 243
12.4.2 Build arguments . . . 243
12.5 Part 4 — Starting and configuring the whole system . . . 244
12.5.1 Why `up-all` runs in two phases . . . 244
12.6 Part 5 — The release process . . . 245
12.6.1 The manual escape hatch . . . 247
12.6.2 Two release failures that are silent . . . 247
12.6.3 A package the workflow did not create . . . 247
12.7 Part 6 — Maintenance burden . . . 248
12.7.1 Adding a sixth application . . . 248
12.7.2 Known duplication, and how to detect drift . . . 248
12.7.3 Standing costs . . . 249
12.8 Part 7 — Troubleshooting . . . 249
12.8.1 Two signals that look like proof and are not . . . 249
12.8.2 Tracking down a network block . . . 250
12.8.3 Other things that will bite . . . 251
12.9 Part 8 — What a genuine multi-host split would still need . . . 252
12.9.1 What exists today . . . 252
12.9.2 What is missing . . . 252
12.10Reference . . . 253

13 Exchange Experiments Roadmap **255**
13.1 How to use this page . . . 255
13.2 Experiment 1: Price-Time Priority Audit Trace . . . 255
13.3 Experiment 2: Exchange Tick-Size Regime per Symbol . . . 256
13.4 Experiment 3: Session-Aware Order Entry Policy Matrix . . . 257
13.5 Experiment 4: Per-Gateway Rate Limits and Burst Controls . . . 258
13.6 Experiment 5: Fee and Rebate Model (Maker-Taker) . . . 259
13.7 Experiment 6: Realistic Cancel/Amend Latency Simulation . . . 260
13.8 Experiment 7: Full Order State Machine with Reason Codes . . . 261
13.9 Experiment 8: Market Data Channel Separation (Level 1 vs Level 2) . . . 262
13.10Experiment 9: User and Account Domain Refactor . . . 263
13.11Experiment 10: Kill Switch and Fat-Finger Controls . . . 264
13.12Experiment 11: Drop Copy and Post-Trade Reconciliation Stream . . . 265
13.13Experiment 12: Matching Mode Plug-in Layer (Pro-Rata vs Price-Time) . . . 266
13.14Experiment 13: Cross-Product Risk and Portfolio Margin Approximation . . . 267
13.15Experiment 14: Auction Imbalance Feed and Indicative Price Model . . . 268
13.16Experiment 15: Synthetic Orders Engine (Parent-Child Execution) . . . 269
13.17Suggested progression path . . . 270
13.18Experiment 16: Self-Trade Prevention . . . 270
13.19Experiment 17: Statistics SQL Query CLI Layer . . . 271
13.20Experiment 18: Drop Copy Durability and Restart Recovery . . . 272
13.21Experiment 19: Cross-Host Operational Validation Matrix . . . 273
13.22Experiment 20: Installed-Mode vs. Source-Mode Parity Validation . . . 274

14 The Release Process **277**
14.1 How a release is produced . . . 277
14.1.1 Publishing by hand . . . 279
14.2 Developer release checklist . . . 279
14.2.1 Before tagging . . . 279
14.2.2 Tag and release . . . 279
14.2.3 After the release . . . 279
14.3 Summary: To make a release, step-by-step . . . 280

14.4 Appendix: The release scripts . . . 280
14.5 What `mkbld.sh` does . . . 280
14.5.1 What `mkrelease.sh` expects . . . 281
14.5.2 What `mkghrelease.sh` expects . . . 281
14.6 Common pitfalls for new developers . . . 281
14.6.1 Forgetting which data directory is live . . . 281
14.6.2 Starting clients before the engine . . . 281
14.6.3 Using a gateway ID that is not in `engine_config.yaml` . . . 281
14.6.4 Forgetting that observer processes depend on each other . . . 281
14.6.5 Assuming docs-only changes need no validation . . . 282
14.6.6 Treating helper scripts as canonical truth . . . 282
14.6.7 Ignoring end-to-end behavior . . . 282
14.7 Suggested first-week path for a new developer . . . 282

CHAPTER 1

How to Use This Book

The Architecture and Developer Guide explains **how EduMatcher is built and why**, and gives contributors what they need to change it safely: the process and message architecture, the matching engine and order book, an order's path through the code, the development workflow, verification, and releases.

Who it is for	Contributors and reviewers, and teachers who use the code as a case study
What it assumes	You have run the exchange (the Quick Start Guide), read Python comfortably, and know the trading basics from the Participant Guide, Part I
What it leaves to other books	Using and operating the system (the Participant and Operator's Guides); protocol formats (Protocols and Clients)

1.1 How the book is organized

Part	What you learn
I. Architecture	Why the system is a set of processes on a message bus, how they are connected, and a guided tour of the whole design
II. Inside the engine	The order book's data structures and matching algorithm, and one order's path through the engine, step by step
III. Developing	Setting up, the development loop, deterministic verification, message generation, containers and networks, and ideas for extensions
IV. Releasing	How a release is produced and published

1.2 Where to start

- **To understand the design:** Architecture, then Order Flow Through the Engine.
- **To make a change:** Development Practice and The Development Loop, then the chapters for the area you are changing.
- **For design decisions:** the design notes in the repository's `docs-design/` directory record the reasoning behind each major feature.

CHAPTER 2

Architecture

Learning objectives

After reading this page you will understand:

- Why a multi-process message-passing architecture was chosen over a monolith or a shared-memory design, and how this mirrors real trading systems
- The fundamental messaging concepts: topics, publish/subscribe, point-to-point, broadcast, and subscription filters
- How the ZeroMQ brokerless topology connects all processes in EduMatcher
- The internal data structures of an order book — heaps, lazy deletion, price-level indexes — and how to read a visual order book depth diagram
- The time complexity of every core operation: insert, match, cancel, stop trigger
- The principal data flows, the single-host/remote deployment boundary, and the trust model
- How state persists and is recovered across a restart, and what is *not* durable
- The runtime failure modes — ZeroMQ back-pressure and silent PUB drop to slow consumers
- Which configuration choices materially change throughput, latency, and memory

This page vs. the Detailed Walkthrough

This page is the **architecture reference**: decisions, data flows, persistence, failure modes, and performance. For a first-principles, code-level tour that builds the same system up from a single limit order, read the Detailed Walkthrough. Where the two overlap, this page is the concise reference and the Walkthrough is the narrative.

2.1 Why This Architecture?

2.1.1 Three ways to build a trading system

Before looking at what EduMatcher does, it helps to understand the alternatives and why they fall short for this use case.

Option 1 — Monolith One process, one program. The user types an order, matching runs, results print. Simple to build, simple to debug. Falls apart as soon as two users need to trade simultaneously, or when you want to add an audit log without touching the matching code.

Option 2 — Shared memory / threads Multiple threads share an in-process order book protected by a lock. Common in C++ HFT cores. Fast, but extremely hard to reason about: lock contention, priority inversion, and data races all wait to bite you. Also single-machine — you can't run the viewer on a separate screen without redesigning everything.

Option 3 — Multi-process message passing Each process owns its own state and communicates by sending messages. No shared memory, no locks. Processes can run on different machines. Adding a new observer (an audit logger, a new data feed) means writing a new subscriber — the engine doesn't change.

EduMatcher uses Option 3. It is also how most real exchange systems are actually built: a matching engine core that publishes every event on an internal bus, with a fleet of downstream consumers (clearing, risk, market data distribution, surveillance) that subscribe to the topics they care about.

2.1.2 Core messaging concepts

Before reading the topology diagram, it is worth having clear definitions:

Topic A string label attached to every message that identifies what kind of event it carries. Examples: `trade.executed`, `order.fill.GW01`, `book.AAPL`. Think of it like the subject line of an email — the sender sets it, the receiver uses it to decide whether to read the body.

Publish / Subscribe (PUB/SUB) One sender, many potential receivers. The sender *publishes* to a topic; every process that has *subscribed* to that topic receives a copy. Processes that have not subscribed never see the message. This is how the engine broadcasts book updates: one `book.AAPL` message is published; the viewer, the stats recorder, and the board all receive it independently.

Subscription filter A subscriber declares which topics it wants to receive. In ZeroMQ, the filter is a simple prefix match: a subscription for `"order.fill.GW01"` receives only messages whose topic starts with that string. A subscription for `"book."` receives book updates for all symbols. An empty-string subscription receives everything.

Broadcast A message sent to all subscribers simultaneously with no specific address. The engine broadcasts `session.state` when the trading phase changes — every process (gateways, viewers, stats, board) reacts according to its own logic.

Point-to-point (PUSH/PULL) One sender, exactly one receiver. EduMatcher uses PUSH/PULL for order submission: a gateway *pushes* an order message and the engine *pulls* it. No other process sees that message; no routing table is needed because the engine is the only process that ever binds the PULL socket.

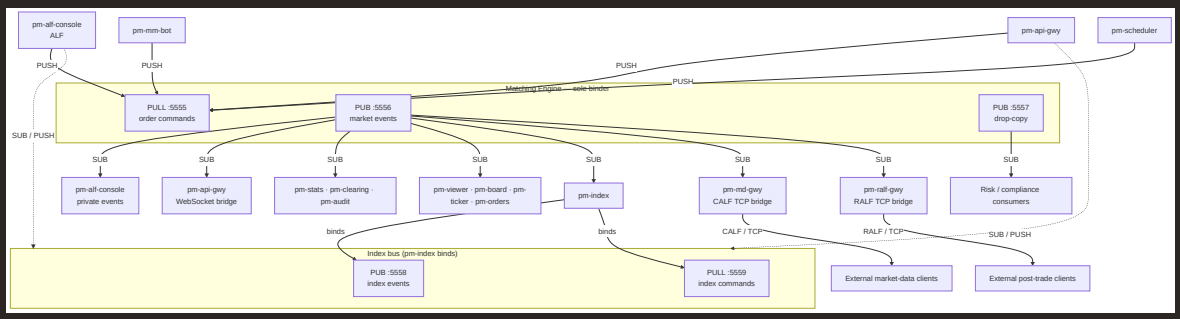
Message routing In a broker-based system, the broker reads the topic and decides which queue to put the message in. In EduMatcher, routing is done by the ZeroMQ layer using the subscription filter — no broker, no routing table, no extra process.

Private reply over PUB/SUB The subscription prefix enables a lightweight request/reply pattern without a dedicated REQ/REP socket. When a gateway pushes a `system.symbols_request` command, the engine publishes the answer on `system.symbols.GW01` — a topic only that gateway's SUB filter will match; every other subscriber silently discards it at the ZMQ layer. This pattern covers every personalised engine response: `order.ack.{GW}`, `system.symbols.{GW}`, `system.quote_bootstrap.{GW}`, and more. It requires every connected process to register a unique gateway ID.

2.2 Overview

EduMatcher uses a **broker-less ZeroMQ** topology.
The matching engine is the only process that *binds* sockets — all other processes *connect* to it.
No ZMQ broker daemon, no message queue server, no external dependencies beyond ZMQ itself.

2.3 ZMQ Topology



2.4 Message Topics

Messages are multipart ZMQ. The first two frames are the message; the frames behind them are the **envelope** — metadata about the message rather than content of it.

Frame	PUB (engine → subscribers)	PUSH (client → engine)
0	topic string (used for SUB filtering)	topic string
1	JSON payload	JSON payload

Frame	PUB (engine → subscribers)	PUSH (client → engine)
2	per-topic sequence number	causal envelope
3	causal envelope	—

Why the envelope is not in the payload. Keeping it in frames means the hot publish path never decodes and re-encodes a message to stamp it, and the same three fields do not have to be declared on each of the 121 message types in `spec/messages/` — and re-declared on every type added later. `decode()` reads only frames 0 and 1, so a consumer that does not care is unaffected.

Frame 2 (PUB) — per-topic sequence. ZeroMQ PUB/SUB drops silently once a subscriber falls behind its high-water mark, and nothing in the delivered message reveals it. A monotonic counter, counted *per topic* so that filtering by topic prefix does not produce phantom gaps, lets any subscriber notice a hole. Read it with `decode_sequence`.

The causal envelope — `msg_id`, `causation_id`, `correlation_id`; see `models/envelope.py` and Causal envelope below. Read it with `decode_envelope` (PUB) or `decode_push_envelope` (PUSH).

2.4.1 Causal envelope

Every message carries a ULID `msg_id`. Every message the engine publishes *while handling an inbound request* also carries that request's `msg_id` as its `causation_id`, and a `correlation_id` naming the message the chain began with.

```
gateway PUSH  order.new                msg=A  cause=—  chain=A
engine  PUB   order.ack.TRADER01       msg=B  cause=A  chain=A
engine  PUB   trade.executed           msg=C  cause=A  chain=A
engine  PUB   order.fill.TRADER01      msg=D  cause=A  chain=A
engine  PUB   order.fill.TRADER02      msg=E  cause=A  chain=A
```

"Which submission produced this trade?" is `cause`. "Everything that flowed from that submission" is one lookup on `chain`. Neither has to be inferred from timing or from shared order ids.

A message with **no** `causation_id` was not caused by anything on the bus — a scheduler tick, a circuit-breaker trip, an end-of-day sweep. That is a positive statement, not missing data.

Two wrappers in `messaging/bus.py` apply this, rather than each publish site remembering to. Both are typed by the `PushSocket` protocol rather than by `zmq.Socket`: a caller holds something that *sends frames and can be closed*, not a raw socket, and saying so in the types is what keeps a wrapper substitutable — by the other wrapper, or by a test double.

- `CausalPusher` — wraps every PUSH socket, stamping a **root** envelope on each client request. A submission is the start of a chain.
- `CausalPublisher` — wraps the engine's PUB socket. The receive loop calls `set_cause(envelope)` with the inbound request before dispatching and clears it in a `finally` afterwards, so attribution cannot leak onto the next message or onto maintenance work.

This is safe precisely because the engine is single-threaded: exactly one inbound message is ever in flight, so there is never more than one cause in scope. A multi-threaded publisher would need a context variable instead.

`pm-audit` records the sequence and the envelope in each log line, between the topic and the payload:

```
[2026-09-11T09:31:02.121+00:00] [order.ack.TRADER01] [seq=7 msg=01J… cause=01J… chain=01J…] {"order_id": "..."}
```

The section is optional, so lines archived before it existed still parse.

2.4.2 Commands — GW / operator → Engine (PUSH :5555)

Topic	Description
`order.new`	Submit a new order
`order.cancel`	Cancel a resting order
`order.amend`	Amend the price and/or quantity of a resting order
`order.combo`	Submit a multi-leg combo order

Topic	Description
order.combo_cancel	Cancel a combo and all its child legs
order.oco	Link two resting orders into an OCO pair
order.oco_cancel	Cancel both legs of an OCO pair
quote.new	Submit or replace a two-sided market-maker quote
quote.cancel	Cancel the active quote for a symbol
risk.kill_switch	Cancel all resting orders and quotes for a gateway
risk.symbol_halt	Operator halt of a single symbol
risk.symbol_resume	Resume a previously halted symbol
risk.cancel_symbol	Cancel all resting orders across all gateways on one symbol
risk.circuit_breaker_halt_all	Administrative global halt
risk.circuit_breaker_resume_all	Administrative global resume
system.gateway_connect	Authenticate a gateway ID on startup
system.gateway_disconnect	Graceful disconnect notice from a gateway
system.symbols_request	Request the list of configured symbols
book.snapshot_request	Request an immediate book snapshot for a symbol
order.orders_request	Request the current resting order list for a gateway
system.quote_bootstrap_request	Request active quote state for a gateway
system.quote_legs_request	Request quote leg snapshot (QLEGS)
system.session_state_request	Request current session state
system.gateways_request	Request the list of configured gateways and connection status
system.volume_request	Request cumulative traded volume for all symbols
system.halt_status_request	Request a snapshot of all currently halted symbols
system.position_request	Request per-symbol position snapshot for a gateway
session.transition	Request a session-phase change (sent by pm-scheduler)

2.4.3 Private replies — Engine → GW (PUB :5556, personalised prefix)

Topic	Description
order.ack.{GW_ID}	Order accepted or rejected
order.fill.{GW_ID}	Partial or full fill notification
order.cancelled.{GW_ID}	Cancel confirmed or SMP-forced cancellation
order.amended.{GW_ID}	Amendment confirmed
order.expired.{GW_ID}	DAY / ATO / ATC order expired at phase change or shutdown
order.orders.{GW_ID}	Response to order.orders_request: full resting order list
combo.ack.{GW_ID}	Combo order accepted or rejected
combo.status.{GW_ID}	Combo lifecycle state change
oco.ack.{GW_ID}	OCO link accepted
oco.cancelled.{GW_ID}	Sibling leg cancelled after other leg filled or was cancelled
quote.ack.{GW_ID}	Quote accepted or rejected
quote.status.{GW_ID}	Quote lifecycle state change

Topic	Description
`risk.kill_switch_ack.{GW_ID}`	Kill-switch execution confirmed
`system.gateway_auth.{GW_ID}`	Authentication accepted or rejected
`system.symbols.{GW_ID}`	Response to `system.symbols_request`
`system.quote_bootstrap.{GW_ID}`	Active quote bootstrap state
`system.quote_legs.{GW_ID}`	Quote leg snapshot response
`system.session_status.{GW_ID}`	Current session state and enforcement flag
`system.gateways.{GW_ID}`	Configured gateways and connection status
`system.volume.{GW_ID}`	Cumulative traded volume per symbol
`system.halt_status.{GW_ID}`	Currently halted symbols snapshot
`system.position.{GW_ID}`	Per-symbol position snapshot (net qty, avg cost)

2.4.4 Broadcasts — Engine → all subscribers (PUB :5556)

Topic	Description
`session.state`	Session phase changed; every subscriber reacts accordingly
`auction.result.{SYMBOL}`	Auction uncross result: equilibrium price, quantity, imbalance
`trade.executed`	A trade was matched; consumed by clearing, stats, RALF, viewers
`book.{SYMBOL}`	Full order-book snapshot. **Throttled**, not per-event: after a state change the snapshot is published on the next engine poll tick, and at most once per `snapshot_interval_sec` (default 0.5 s) per symbol. On-demand `book.snapshot_request` replies are immediate. See Configuration choices that impact performance.
`depth.{SYMBOL}`	Depth-of-market statistics (bid/ask imbalance, cost-to-move) published on the **same throttled tick** as `book.{SYMBOL}`.
`circuit_breaker.halt.{SYMBOL}`	Symbol halted by the circuit breaker
`circuit_breaker.resume.{SYMBOL}`	Symbol resumed after a circuit-breaker halt
`system.eod`	End-of-day shutdown broadcast; signals all subscribers to flush and stop

2.4.5 Drop-copy feed — Engine → compliance consumers (PUB :5557)

Topic	Description
`drop_copy.event.{GW_ID}`	Sequenced fill event with nanosecond timestamp, one per filled order leg
`drop_copy.replay.{CLIENT_ID}`	Replayed historical fill events in response to a replay request

2.4.6 Index bus — pm-index clients (PUB :5558 / PULL :5559)

`pm-index` is its own process with its own dedicated sockets at ports :5558 and :5559. It subscribes to the engine PUB at :5556 and publishes index events independently.

Topic	Direction	Description
`index.update`	pm-index → all	Current index level, OHLC, and session state — also consumed live by pm-stats for level/EOD history

Topic	Direction	Description
`index.history_request`	GW → pm-index	Query structural/audit history (`INIT`, `CORP_ACTION`, `ADD_CONSTITUENT`, `DELIST`) by time range — not level/EOD data, which lives in pm-stats' SQLite tables
`index.history.{GW_ID}`	pm-index → GW	Structural/audit history query response
`index.corp_action`	GW → pm-index	Apply a corporate action to an index constituent
`index.corp_action_ack.{GW_ID}`	pm-index → GW	Corporate action acknowledgement
`index.constituent_change`	GW → pm-index	Add or remove an index constituent
`index.constituent_change_ack.{GW_ID}`	pm-index → GW	Constituent-change acknowledgement

2.4.7 Market-model bus — pm-market-sim clients (PUB :5553 / PULL :5554)

`pm-market-sim` binds its own pair of sockets, on loopback by default (`EDUMATCHER_SIM_BIND_HOST`). It follows the engine's session from :5556 and publishes the market model: true values for the AI traders, ADMIN and `pm-mm-bot --anchor-sim`, and headlines for everyone. See The Market Model.

Topic	Direction	Description
`sim.value`	pm-market-sim → all	Every symbol's true value, once per step during continuous trading
`sim.state`	pm-market-sim → all	Heartbeat: `RUNNING` or `PAUSED`, the session, the step
`news.event`	pm-market-sim → all	One headline, or a rumour confirmed or retracted
`sim.command`	ADMIN → pm-market-sim	Status request; inject, confirm or retract news (`pm-news`)
`sim.command_ack.{GW_ID}`	pm-market-sim → ADMIN	Command acknowledgement

2.5 Process Roles

Process	ZMQ Sockets	Binds/Connects	Role
pm-engine	PULL :5555, PUB :5556, PUB :5557	**Binds** all three	Matching, session state, combo/OCO/quote tracking, drop-copy
pm-alf-console (ALF)	PUSH→:5555, SUB→:5556, SUB→:5558, PUSH→:5559	Connects	Interactive order entry; ALF line protocol; index data display
pm-api-gwy	PUSH→:5555, SUB→:5556, SUB→:5558, PUSH→:5559	Connects	REST and WebSocket order gateway for programmatic clients

Process	ZMQ Sockets	Binds/Connects	Role
pm-mm-bot	PUSH→:5555, SUB→:5556	Connects	Automated market-maker; manages two-sided quotes
pm-scheduler	PUSH→:5555	Connects	Drives session phase transitions on a time schedule
pm-viewer	SUB→:5556	Connects	Real-time order book display for one symbol
pm-orders	SUB→:5556	Connects	Order status monitor across all gateways
pm-audit	SUB→:5556, REP :5562	**Binds** :5562 (health); connects→:5556	Universal event log (subscribes to all topics)
pm-clearing	SUB→:5556, REP :5563	**Binds** :5563 (health); connects→:5556	P&L tracking and trade settlement
pm-stats	SUB→:5556, SUB→:5558, PUSH→:5555, REP :5561	**Binds** :5561 (health); connects→:5555, :5556, :5558	OHLCV statistics and SQLite persistence
pm-ticker	SUB→:5556, SUB→:5553	Connects	Scrolling market data display, with the latest headline
pm-board	SUB→:5556	Connects	Multi-symbol paged market display
pm-index	PULL :5559, PUB :5558, REP :5564, SUB→:5556	**Binds** :5558, :5559 and :5564 (health); connects→:5556	Index calculation, OHLC, corporate actions
pm-market-sim	PUB :5553, PULL :5554, PUSH→:5555, SUB→:5556	**Binds** :5553 and :5554; connects→:5555, :5556	Market model: true values and news
pm-ai-trader / pm-ai-swarm	SUB→:5556, SUB→:5553, PUSH→:5555; ALF TCP to pm-alf-gwy	Connects	AI traders; orders go through `pm-alf-gwy`, only read requests to :5555

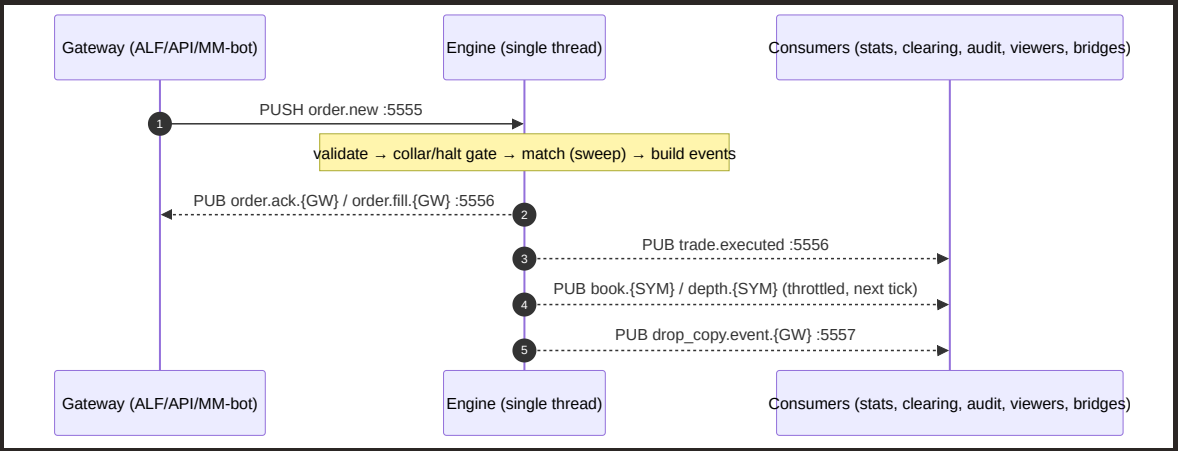
Process	ZMQ Sockets	Binds/Connects	Role
pm-md-gwy (CALF)	SUB→:5556, SUB→:5558	Connects	Translates engine events to CALF TCP for external market-data subscribers
pm-ralf-gwy (RALF)	SUB→:5556	Connects	Translates trade events to RALF TCP for external post-trade / clearing parties
pm-monitor	PULL :5610, PUB :5611, SUB→:5556	**Binds** :5610 and :5611; connects→:5556	Health collector: judges every process from its beacon, computes rates and thresholds, keeps `metrics.db`, publishes `health.snapshot/heal`

Every long-running process above (not the interactive viewers and consoles) also has a `HealthReporter`: a background thread with a PUSH→:5610 socket that sends a `health.beacon` to `pm-monitor` every two seconds — memory, CPU, how long ago its main loop last completed an iteration, and its counters. Sent from a thread rather than the loop, so a hung loop still reports (with a growing loop age), and sent without blocking, so a missing `pm-monitor` costs nothing. See Health Monitoring.

2.6 Data Flows

Four principal flows move through the system. All of them converge on the single-writer engine and fan back out over PUB topics.

1. Order entry (request/response). A gateway PUSHes a command on `:5555`; the engine processes it run-to-completion and publishes personalised results on the gateway's private `order.*` / `quote.*` / `system.*` prefixes on `:5556`.



2. Market data (broadcast, throttled). After matching, `trade.executed` is published immediately, but `book.{SYM}` / `depth.{SYM}` snapshots are coalesced and emitted on the poll tick, at most once per `snapshot_interval_sec` per symbol

(see Configuration choices that impact performance). Protocol bridges (`pm-md-gwy`/CALF) re-publish these to external TCP subscribers.

3. Post-trade & drop-copy. `trade.executed` feeds `pm-clearing` (P&L), `pm-stats` (OHLCV), and `pm-audit` (full log) on `:5556`; the sequenced per-fill drop-copy feed on `:5557` feeds compliance consumers and `pm-ralf-gwy` (RALF).

4. Index sub-bus. `pm-index` is a *second-tier publisher*: it subscribes to the engine's `:5556`, recomputes index levels, and publishes `index.update` on its own bus (`:5558`), accepting commands on `:5559`. It and `pm-market-sim` (`:5553`, `:5554`) are the only processes besides the engine that bind ZeroMQ sockets.

Where each of these ends up on disk is covered in Persistence & crash recovery; how they behave when a consumer can't keep up is covered in Failure modes & fault tolerance.

2.7 Key Design Decisions

2.7.1 The Engine as the Sole Binder

The engine is the only process that binds sockets (:5555, :5556, :5557). Every other process connects to the engine. This has three practical consequences:

- **Start the engine first.** A gateway that connects before the engine is started will block on send until the engine comes up (ZMQ queues outgoing messages).
- **Adding a new subscriber requires no engine change.** Any process can connect to :5556 and subscribe to the topics it needs. The engine has no knowledge of who is listening.
- **Restart isolation.** Any subscriber can crash and restart without disturbing the engine or other subscribers. It simply reconnects and resumes.

`pm-index` and `pm-market-sim` are the exceptions: each binds its own pair of sockets (:5558/:5559 and :5553/:5554) because it is a second-tier publisher, not a consumer of the engine.

2.7.2 Two Tiers of Gateways

EduMatcher has two distinct gateway roles:

Order-entry gateways (`pm-alf-console`, `pm-api-gwy`, `pm-mm-bot`) connect directly to the engine over ZMQ PUSH/PULL. They authenticate via `system.gateway_connect` and receive personalised events on their subscribed `order.*`, `quote.*`, and `system.*` topics. These processes are internal to the exchange operator.

Protocol bridges (`pm-md-gwy`, `pm-ralf-gwy`) subscribe to the engine PUB socket and translate engine events into an external TCP line protocol for third-party consumers:

- `pm-md-gwy` (CALF) — real-time order book and trade data for market-data subscribers, with session management, subscription filtering, and gap-recovery replay
- `pm-ralf-gwy` (RALF) — post-trade execution events for external clearing, drop-copy, and audit parties over authenticated TCP sessions

2.7.3 Bus Segmentation by Function

Separating traffic onto distinct ports serves both operational and compliance needs:

Port	Socket	Purpose
:5555	PULL	Order commands — only the engine reads these
:5556	PUB	Market events — all internal subscribers
:5557	PUB	Drop-copy — sequenced fill feed for compliance consumers
:5558	PUB	Index events — separate bus owned by `pm-index`
:5559	PULL	Index commands — received only by `pm-index`

The drop-copy feed on :5557 deliberately isolates the compliance feed from the full market event stream. Each fill event carries a monotonic sequence number and a nanosecond timestamp so consumers can detect gaps.

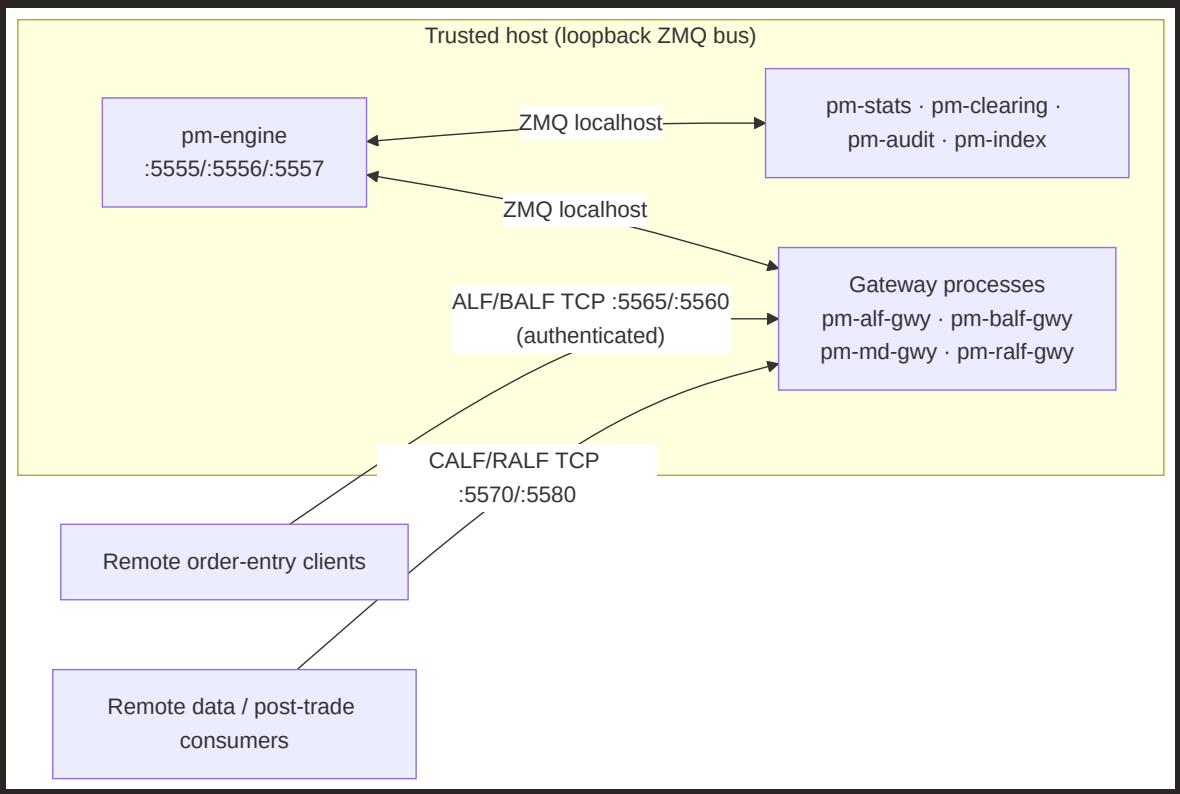
2.7.4 Single-Threaded Engine as a Correctness Guarantee

The engine's main loop processes one message at a time in a single thread. There are no locks, no shared mutable state, and no concurrent modifications. Combo status transitions, cascade-cancels, and session phase changes are all fully serialised. The price is that a long auction uncross blocks all other message processing until it finishes — acceptable for an educational system, but a production engine would decompose the critical path or use non-blocking event dispatch.

2.7.5 Deployment Topology & Trust Boundaries

EduMatcher is **partially multi-host by design**. The core is single-host; remote reach is provided only through the external protocol gateways.

- **The engine is never partitioned.** There is exactly one authoritative matching engine. It cannot be sharded or run active/active — a second engine would be a second, divergent order book.
- **The internal ZMQ bus is loopback by default.** The engine binds `tcp://127.0.0.1` for `:5555/:5556/:5557` (and `pm-index` for `:5558/:5559`). These are **unauthenticated**: the PUSH socket accepts any `gateway_id` in the payload, and `ADMIN` is a payload-level role check, not transport authentication. The bus therefore MUST stay on a trusted host. The internal consumers (`pm-stats`, `pm-clearing`, `pm-audit`, viewers, `pm-index`, and the protocol-gateway processes) are expected to be **co-located** with the engine.
- **Remote participants connect at the gateways, not the bus.** The protocol gateways bind `0.0.0.0` and are the intended network edge: order entry over **ALF** (`pm-alf-gwy`, :5565) and **BALF** (`pm-balf-gwy`, :5560), which authenticate sessions against the `participants` allowlist; market data over **CALF** (`pm-md-gwy`, :5570) and post-trade over **RALF** (`pm-ralf-gwy`, :5580). A remote trading client or data consumer speaks TCP to one of these — it never touches the ZMQ bus.



Listeners fall into two planes with different defaults:

	Core plane	Service layer
Processes	`pm-engine`, `pm-index`	the four protocol gateways, `pm-log-srv`, `pm-api-gwy`

	Core plane	Service layer
Default bind host	`127.0.0.1`	`0.0.0.0`
Override	`EDUMATCHER_ENGINE_BIND_HOST`, `EDUMATCHER_INDEX_BIND_HOST`	`EDUMATCHER_GATEWAY_BIND_HOST`, then `bind_address:` / `host:`

The core plane is an internal ZMQ bus between co-located processes, so it defaults to loopback. Widening it *can* place ZMQ consumers on another machine — but doing so exposes the **unauthenticated** bus over the network and SHOULD only be done on a private, trusted segment. The supported remote path is the gateways, which default to `0.0.0.0` because being reachable is their purpose; the ALF and BALF order-entry gateways authenticate, the read-only feeds do not.

2.8 Order Book Data Structures

2.8.1 Visual: order depth at each price level

Before looking at the heap internals, here is what an order book actually looks like from a market participant's perspective. Each price level accumulates all the resting orders placed at that price. Time priority determines which order fills first within a level.

```
       BID SIDE                          ASK SIDE
  (buyers waiting to buy)           (sellers waiting to sell)

  Price    Qty   Orders              Price    Qty   Orders
  ──────  ─────  ──────              ──────  ─────  ──────
  150.00   800     3    ◄── best bid  150.25   500     1    ◄── best ask
  149.75   650     2                  150.50   300     2
  149.50   200     1                  150.75   450     3
  149.25   500     2                  151.00   200     1
  149.00   300     1                  151.25   350     2

                   ▲                   ▲
                   └── SPREAD ─────────┘
                      (150.25 - 150.00 = 0.25)
```

Reading this table:

- The **best bid** (150.00) and **best ask** (150.25) define the current spread.
- A MARKET BUY would consume the ask side top-down (best ask first): 500 @ 150.25, then 300 @ 150.50, etc., until its quantity is filled.
- A MARKET SELL would consume the bid side top-down: 800 @ 150.00, then 650 @ 149.75, etc.
- A LIMIT BUY at 150.10 would *not* cross the spread (best ask is 150.25 > 150.10), so it would rest on the bid side at a new level between 150.00 and 150.25.
- Within the 150.00 bid level, 3 separate orders are resting. The order with the earliest timestamp fills first when a sell aggressor arrives.

The "Qty" column shows **total visible quantity** at that level. Iceberg orders only contribute their `visible_qty` (the current peak); the hidden reserve is not shown.

2.8.2 Internal representation

```
OrderBook (per symbol)
├── _bids        max-heap  [(-price, timestamp, order), ...]
├── _asks        min-heap  [( price, timestamp, order), ...]
├── _buy_stops   min-heap  [( stop_price, timestamp, order), ...]
├── _sell_stops   max-heap  [(-stop_price, timestamp, order), ...]
├── _order_index  dict[order_id → Order]   (all resting orders)
└── _entry_index  dict[order_id → HeapEntry] (bid/ask heap entries)
```

Price-time priority: within the same price level, earlier-submitted orders are filled first. Lazy deletion is used — heap entries are marked invalid on cancel/fill and skipped on next access.

2.9 Thread Model

Each process is **single-process, single-thread** for the main logic, with one optional background thread for ZMQ receiving in interactive processes (gateway, viewer, orders, clearing).

The engine runs a single-threaded event loop using `zmq.Poller` with a 200 ms timeout, making it safe from concurrent modification without locks.

2.10 Core Matching Algorithm — In Depth

This section describes the exact data structures, algorithms, and time complexities used by the matching engine.

2.10.1 Order Book Organization

Each symbol gets its own `OrderBook` instance. Internally it maintains **eight primary data structures** organized for fast price-time-priority matching:

```
OrderBook("AAPL")
│
├── _bids          max-heap  [HeapEntry(-price, timestamp, order), ...]
├── _asks          min-heap  [HeapEntry( price, timestamp, order), ...]
│
├── _buy_stops     min-heap  [(stop_price, timestamp, order), ...]
├── _sell_stops    max-heap  [(-stop_price, timestamp, order), ...]
│
├── _bid_qty       dict[int, int]      price_ticks → total visible resting qty
├── _ask_qty       dict[int, int]      price_ticks → total visible resting qty
│
├── _order_index   dict[order_id, Order]       all resting orders (fast cancel lookup)
└── _entry_index   dict[order_id, HeapEntry]   heap entry pointers (lazy delete)
```

Why heaps? Python's `heapq` gives us O(log n) insertion and O(1) peek at best price. Since we always match against the best available price, a heap is the natural choice.

Bids use negated prices so that `heapq` (a min-heap) pops the *highest* bid first:

```
# Bid key:  (-price, timestamp)  →  highest price wins, ties broken by earliest time
# Ask key:  ( price, timestamp)  →  lowest  price wins, ties broken by earliest time
```

Price-level quantity indexes (`_bid_qty`, `_ask_qty`) are auxiliary `dict[price, int]` maps that track the aggregate visible quantity at each price level. They enable O(p) FOK pre-checks (where p = number of distinct price levels) instead of walking every heap entry.

2.10.2 Heap Entry and Lazy Deletion

Each heap entry is a wrapper object:

```
@dataclass
class HeapEntry:
    key:   tuple       # (-price, ts) for bids; (price, ts) for asks
    order: Order
    valid: bool = True # set False on cancel/fill → "tombstone"
```

When an order is cancelled or filled, we do **not** remove it from the heap immediately (which would require O(n) search + O(log n) sift). Instead we mark `entry.valid = False` (O(1)). Stale entries are garbage-collected lazily when they bubble to the top during `_peek()`:

```
┌──────────────────────────────────────────────────┐
│              _asks min-heap                      │
│                                                  │
│   top → [100.0, t=1, VALID]  ← best ask          │
│         [100.0, t=3, INVALID]  ← tombstone, skipped   │
│         [101.5, t=2, VALID]                      │
│         [102.0, t=4, VALID]                      │
└──────────────────────────────────────────────────┘

_peek() pops invalid entries until a valid one is at the top.
Amortized O(1) access to the best price.
```

2.10.3 Matching Algorithm: The Sweep

The core of the matching engine is the `_sweep()` function. It is called by MARKET, LIMIT, FOK, and ICEBERG order types.

Pseudocode

```
function SWEEP(aggressor, opposite_heap, price_limit):
    while aggressor.remaining_qty > 0:
        best ← PEEK(opposite_heap)        // O(1) amortized (lazy GC)
        if best is None:
            break                          // no more resting orders
        if price_limit exists:
            if BUY  and best.price > price_limit: break
            if SELL and best.price < price_limit: break

        // Self-match prevention (SMP)
        if aggressor.gateway_id == best.gateway_id and SMP enabled:
            handle SMP action (cancel aggressor / resting / both)
            continue or return

        fill_qty   ← min(aggressor.remaining_qty, best.remaining_qty)
        fill_price ← best.price            // passive price wins (maker gets their price)

        APPLY_FILL(aggressor, best, fill_qty, fill_price)
```

Visual Flow

```
                Incoming BUY LIMIT @ 101.0, qty=25
                            │
                            ▼
          ┌──────── ASKS HEAP (min-heap) ──────────┐
          │                                        │
          │  [100.0, t=1, qty=10]  ← best ask      │ ← fills 10 @ 100.0
          │  [100.5, t=2, qty=8 ]  ← next best     │ ← fills 8  @ 100.5
          │  [101.0, t=3, qty=20]  ← crosses       │ ← fills 7  @ 101.0
          │  [102.0, t=4, qty=5 ]  ← above limit   │ ← STOP: price > 101.0
          │                                        │
          └────────────────────────────────────────┘
                            │
                            ▼
          Result: 3 trades (10+8+7 = 25 filled), aggressor FILLED
```

Order-Type Dispatch

```
incoming order
      |
      ├─ MARKET  ──→ SWEEP(no price_limit) → discard unfilled remainder
      |
      ├─ LIMIT   ──→ SWEEP(price=order.price) → REST unfilled portion on own side
      |
      ├─ FOK     ──→ PRE-CHECK available qty via _bid_qty/_ask_qty
      |                 if insufficient → REJECT immediately
      |                 else → SWEEP(price=order.price)
      |
      ├─ ICEBERG ──→ SWEEP visible slice → replenish peak from hidden qty → REST
      |
      └─ STOP / STOP_LIMIT ──→ add to stop heap → no immediate match
                                triggers later when last_trade_price crosses stop_price
```

2.10.4 Apply Fill

When a match is found, `_apply_fill` performs these updates atomically:

```
function APPLY_FILL(aggressor, passive, fill_qty, fill_price):
    1. Create Trade object (symbol, buyer/seller, price, qty, timestamp)
    2. Update last_trade_price, last_buy/sell_price
    3. aggressor.remaining_qty -= fill_qty
       → status = FILLED if 0 else PARTIAL
    4. passive.remaining_qty   -= fill_qty
       → status = FILLED if 0 else PARTIAL
       → if FILLED: mark HeapEntry.valid = False (tombstone)
    5. Deduct fill_qty from price-level qty index (_bid_qty / _ask_qty)
    6. If passive is ICEBERG and displayed_qty exhausted:
       → replenish displayed_qty from hidden remainder
       → update timestamp (loses time priority — back of queue)
       → re-insert into heap with fresh key
```

2.10.5 Stop Order Trigger Mechanism

Stop orders live in **separate heaps**, sorted by trigger price:

```
_buy_stops:  min-heap by ( stop_price, timestamp) — triggers when price RISES to/above
_sell_stops: max-heap by (-stop_price, timestamp) — triggers when price FALLS to/below
```

After every trade, `_check_stops()` peeks at both heaps:

```
function CHECK_STOPS(last_trade_price):
    triggered = []

    // BUY stops: sorted cheapest first; fire all where last_price >= stop
    while _buy_stops not empty:
        if top.stop_price > last_trade_price: break
        pop entry
        convert STOP → MARKET (or STOP_LIMIT → LIMIT)
        triggered.append(order)

    // SELL stops: sorted most expensive first; fire all where last_price <= stop
    while _sell_stops not empty:
        if top.stop_price < last_trade_price: break
        pop entry
        convert STOP → MARKET (or STOP_LIMIT → LIMIT)
        triggered.append(order)

    // Re-process each triggered order through the book
    for order in triggered:
        process(order)  → may produce additional trades → may trigger more stops (recursion)
```

2.10.6 Resting an Order

When a LIMIT order does not fully cross the spread, its remainder is placed on the book:

```
function REST(order):
    if BUY:
        key = (-order.price, order.timestamp)   # negated → max-heap behavior
        heappush(_bids, HeapEntry(key, order))
        _bid_qty[order.price] += order.remaining_qty
    else:
        key = (order.price, order.timestamp)
        heappush(_asks, HeapEntry(key, order))
        _ask_qty[order.price] += order.remaining_qty

    _order_index[order.id] = order
    _entry_index[order.id] = entry
```

2.10.7 Cancellation

```
function CANCEL(order_id):
    order = _order_index[order_id]          // O(1) lookup
    entry = _entry_index[order_id]          // O(1) lookup
    entry.valid = False                     // tombstone — O(1)
    deduct remaining qty from _bid_qty or _ask_qty
    order.status = CANCELLED
    return order
```

No heap restructuring needed — lazy deletion handles cleanup on next `_peek()`.

2.10.8 Time Complexity Summary

Operation	Complexity	Notes
Insert (rest on book)	O(log n)	`heapq.heappush`
Best-price access	O(1) amortized	`_peek()` with lazy GC of tombstones
Match one level	O(log n)	Pop from heap
Full sweep (k fills)	O(k log n)	k = number of resting orders matched
Cancel	O(1)	Tombstone + dict lookup
FOK pre-check	O(p)	p = distinct price levels (via qty index)
Stop trigger check	O(k log s)	k triggered stops out of s total
Snapshot (book image)	O(n)	Walk all valid entries
Order lookup by ID	O(1)	`_order_index` dict

Where **n** = total resting orders on one side of one book.

2.10.9 Combo Orders — Data Structures and Tracking

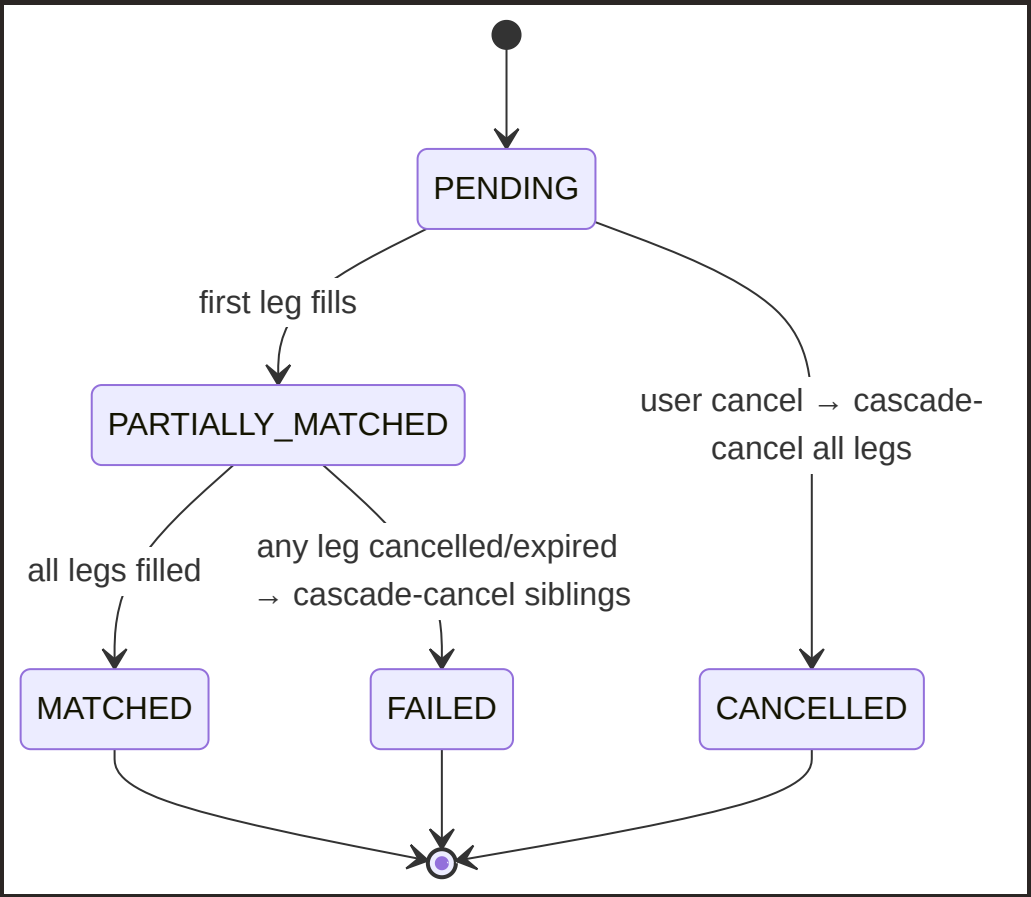
Combo orders are **parent containers** that decompose into normal child orders. The engine uses two dictionaries to track the parent-child relationship:

```
Engine
├── _combos           dict[combo_internal_id → ComboOrder]
└── _order_to_combo   dict[child_order_id → combo_internal_id]
```

A `ComboOrder` holds per-leg state:

```
ComboOrder
├── id                str (internal UUID)
├── combo_id          str (user label)
├── gateway_id        str
├── combo_type        AON
├── tif               DAY | GTC
├── legs              list[ComboLeg]        (2–10 entries)
├── status            PENDING | PARTIALLY_MATCHED | MATCHED | FAILED | CANCELLED
├── child_order_ids   list[str]             (parallel to legs by index)
├── leg_fill_qty      dict[leg_index → int] (filled qty per leg)
└── leg_statuses      dict[leg_index → str] (OrderStatus.value per leg)
```

Combo Lifecycle State Machine



Combo Entry Algorithm

```
function HANDLE_COMBO_ORDER(payload):
    combo = ComboOrder.from_dict(payload)

    // === Validation phase ===
    validate gateway auth                          O(1)
    validate 2 ≤ legs ≤ 10                         O(1)
    validate no duplicate symbols                  O(L) where L=leg count
    validate all symbols in allowlist              O(L)
    validate each leg (qty > 0, price if needed)   O(L)

    ACK combo to gateway

    // === Child order creation phase ===
    for i, leg in enumerate(combo.legs):           O(L)
        child = Order.create(from leg fields)
        child.combo_parent_id = combo.id
        child.leg_index = i

        combo.child_order_ids.append(child.id)     O(1)
        _order_to_combo[child.id] = combo.id       O(1)
        _order_symbol[child.id]   = leg.symbol     O(1)

        trades, events = book.process(child)       O(k log n) per leg

        // Publish fills/rejects for immediate matches
        for event in events:
            publish fill/reject messages

        combo.leg_statuses[i] = child.status.value
        combo.leg_fill_qty[i] = filled amount

    _combos[combo.id] = combo                      O(1)
    UPDATE_COMBO_STATUS(combo)                     O(L)
```

Combo Status Update (after any child event)

```
function CHECK_COMBO_AFTER_CHILD_EVENT(child_order):
    combo_id = _order_to_combo[child_order.id]     O(1)
    combo    = _combos[combo_id]                   O(1)

    idx = child_order.leg_index
    combo.leg_statuses[idx] = child_order.status
    combo.leg_fill_qty[idx] = filled amount

    if child_order.status in (CANCELLED, EXPIRED):
        CASCADE_CANCEL(combo, FAILED)              O(L)
        return

    UPDATE_COMBO_STATUS(combo)                     O(L)

function UPDATE_COMBO_STATUS(combo):
    if all leg_statuses == FILLED:                 O(L)
        combo.status = MATCHED
        publish combo.status MATCHED
    elif any leg has PARTIAL or FILLED fill:
        combo.status = PARTIALLY_MATCHED
        publish combo.status PARTIALLY_MATCHED
```

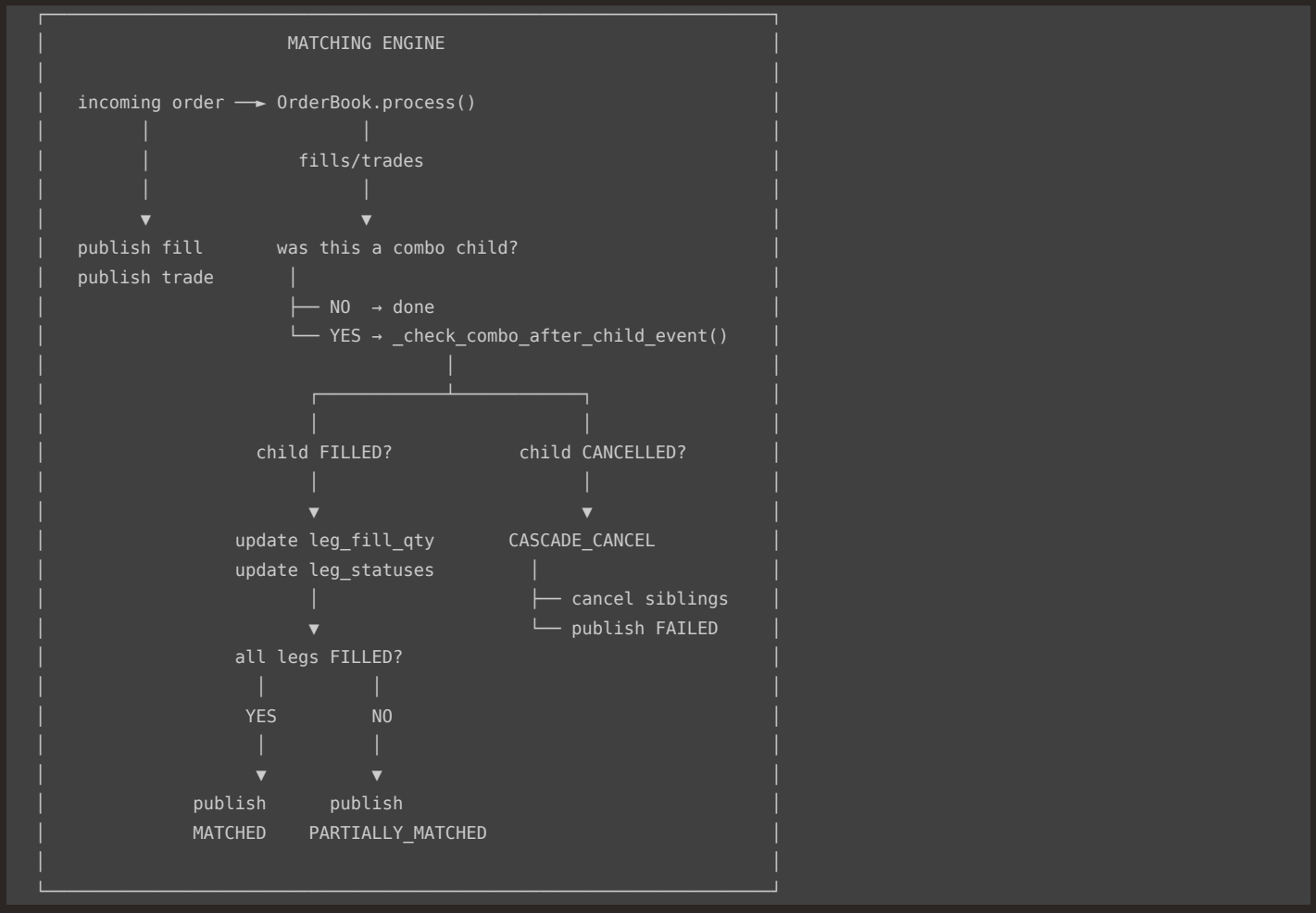
Cascade Cancel

```
function CASCADE_CANCEL(combo, terminal_status):
    combo.status = terminal_status

    for child_id in combo.child_order_ids:          O(L)
        symbol = _order_symbol[child_id]            O(1)
        book   = books[symbol]                      O(1)
        book.cancel_order(child_id)                 O(1) — tombstone
        publish order.cancelled
        remove from _order_symbol, _order_to_combo  O(1)

    publish combo.status
```

Combo Event Propagation Flow



Combo Time Complexity Summary

Operation	Complexity	Notes
Validate combo	O(L)	L = leg count (2–10, bounded constant)
Create & post all children	O(L × k log n)	k matches per leg
Lookup child→combo parent	O(1)	`_order_to_combo` dict
Update combo status	O(L)	Iterate leg_statuses
Cascade-cancel	O(L)	One tombstone per child
Combo cancel by user	O(L)	Lookup by combo_id is O(C) worst-case*

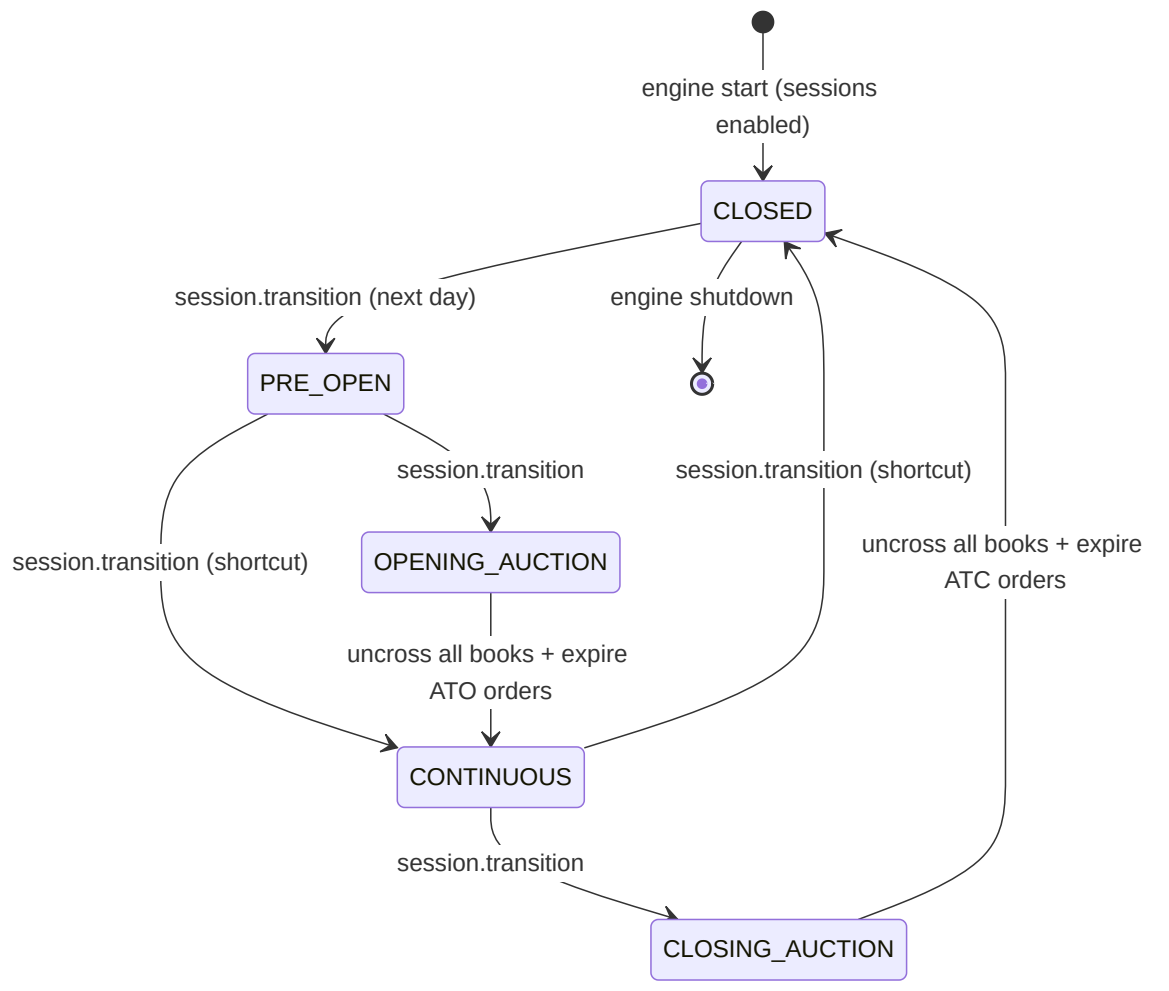
* User-facing cancel searches `_combos` by `combo_id` string (not internal UUID). With C active combos, worst-case is O(C). In practice C is small and could be indexed if needed.

Since L is bounded at 10, all combo-specific operations are effectively **O(1)** in big-O terms relative to book size n.

2.10.10 Session State Machine

The engine manages a session state that controls which order types are accepted and whether matching occurs. The scheduler process drives transitions by sending `session.transition` messages.

States and Transitions



Phase Behavior

Phase	Matching?	Accepts	Rejects
PRE_OPEN	No	LIMIT, STOP, STOP_LIMIT, ICEBERG	MARKET, FOK, IOC, ATO, ATC
OPENING_AUCTION	No	Same as PRE_OPEN + ATO	MARKET, FOK, IOC, ATC
CONTINUOUS	Yes	All types	ATO, ATC
CLOSING_AUCTION	No	Same as PRE_OPEN + ATC	MARKET, FOK, IOC, ATO
CLOSED	—	Nothing	All

During no-matching phases, accepted orders rest on the book but the sweep is never called. Stop orders are stored but do not fire (no trades occur to trigger them).

Handling a Transition

```
function HANDLE_SESSION_TRANSITION(to_state):
    if transition not in VALID_TRANSITIONS[current_state]:
        log warning, ignore
        return

    prev_state = current_state
    current_state = to_state

    // If exiting an auction phase → uncross all books
    if prev_state in (OPENING_AUCTION, CLOSING_AUCTION):
        for each symbol book:
            UNCROSS(book)

    // Expire phase-specific orders
    if prev_state == OPENING_AUCTION:
        expire all ATO orders → publish order.expired
    if prev_state == CLOSING_AUCTION:
        expire all ATC orders → publish order.expired

    publish session.state { state, prev_state }
```

2.10.11 Auction Uncross Algorithm — Equilibrium Price

When exiting an auction phase, accumulated orders execute at a **single equilibrium price**. This is the price that maximizes total traded quantity.

Algorithm

```
function UNCROSS(book):
    // 1. Collect all candidate prices (every distinct bid and ask price)
    candidates = sorted(unique(bid_prices ∪ ask_prices))

    best_price    = None
    best_exec_qty = 0
    best_surplus  = ∞

    // 2. Evaluate each candidate
    for P in candidates:
        buy_qty  = Σ resting bid qty where bid_price ≥ P
        sell_qty = Σ resting ask qty where ask_price ≤ P
        exec_qty = min(buy_qty, sell_qty)
        surplus  = |buy_qty − sell_qty|

        // 3. Selection: maximize exec_qty, then minimize surplus, then lowest price
        if exec_qty > best_exec_qty:
            best_price, best_exec_qty, best_surplus = P, exec_qty, surplus
        elif exec_qty == best_exec_qty and surplus < best_surplus:
            best_price, best_exec_qty, best_surplus = P, exec_qty, surplus
        elif exec_qty == best_exec_qty and surplus == best_surplus and P < best_price:
            best_price = P

    if best_exec_qty == 0:
        publish auction.result { eq_price: null, eq_qty: 0 }
        return

    // 4. Execute: fill orders at equilibrium price using price-time priority
    remaining = best_exec_qty
    while remaining > 0:
        // match best bid against best ask, fill_price = best_price
        best_bid = PEEK(bids)
        best_ask = PEEK(asks)
        fill_qty = min(best_bid.remaining, best_ask.remaining, remaining)
        APPLY_FILL(best_bid, best_ask, fill_qty, best_price)
        remaining -= fill_qty

    publish auction.result { eq_price, eq_qty, trades_count, imbalance_side, imbalance_qty }
```

Key difference from continuous matching: In continuous mode, each fill happens at the **resting** order's price (price improvement for the aggressor). In auction uncross, ALL fills happen at the same computed equilibrium price — neither the bid's limit nor the ask's limit is used directly.

Self-match prevention. The pseudo-code shows the plain curves. When a gateway has a policy and both a bid and an ask in the crossing range, the executable quantity at a price is that of the walk itself: a pair of one gateway is cancelled by the bid's SMP policy (the ask's when the bid's is `NONE`) instead of trading. After the fills, the same rule is applied once more to any bid and ask still crossed (a pair lying beyond the equilibrium price that SMP kept apart), so no auction ends with a crossed book.

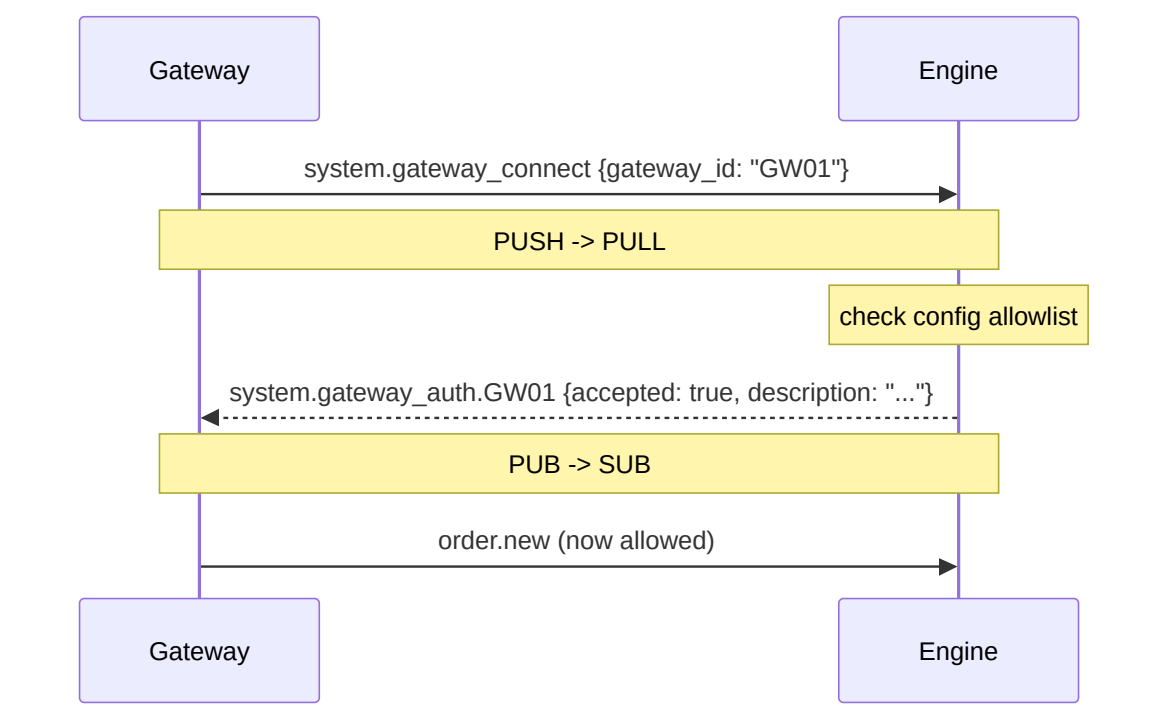
Complexity

Operation	Complexity	Notes
Collect candidates	O(n)	Scan all resting orders
Evaluate all candidates	O(p × n)	p prices × n cumulative sums (optimizable to O(n) with prefix sums)
Execute fills	O(k log n)	k fills at equilibrium price

Where p = distinct price levels, n = total resting orders, k = orders matched.

2.10.12 Gateway Authentication

Before a gateway can submit orders, it must authenticate with the engine. If the engine has a `participants` section in its config, only listed gateway IDs are accepted.



If `accepted: false`, the gateway prints the rejection reason and exits. If no `participants` section exists in config, all gateway IDs are auto-accepted (backward-compatible mode).

Orders from gateways that have not completed the auth handshake are rejected with reason "Gateway not connected: {GW_ID}".

2.10.13 Engine Event Loop

The engine processes messages sequentially in a single thread:

```
function RUN():
    restore_gtc()          // reload GTC orders + combos from disk
    load_config()          // seed stats, inject MM orders

    loop:
        poll PULL socket (200 ms timeout)

        if message available:
            topic, payload = decode(message)
            dispatch:
                "order.new"              → _handle_new_order(payload)
                "order.cancel"           → _handle_cancel(payload)
                "order.combo"            → _handle_combo_order(payload)
                "order.combo_cancel"     → _handle_combo_cancel(payload)
                "session.transition"     → _handle_session_transition(payload)
                "system.gateway_connect" → _handle_gateway_connect(payload)
                "system.symbols_request" → _handle_symbols_request(payload)
                "book.snapshot_request"  → _handle_snapshot_request(payload)
                "order.orders_request"   → _handle_orders_request(payload)

        flush_snapshots()  // publish throttled book images for dirty symbols

        if shutdown requested:
            _shutdown()    // save GTC, expire DAY, save combos, publish EOD
            break
```

No locks, no shared memory, no race conditions. The sequential dispatch guarantees that combo status transitions and cascade-cancels are atomic from the system's perspective.

2.11 Persistence & Crash Recovery

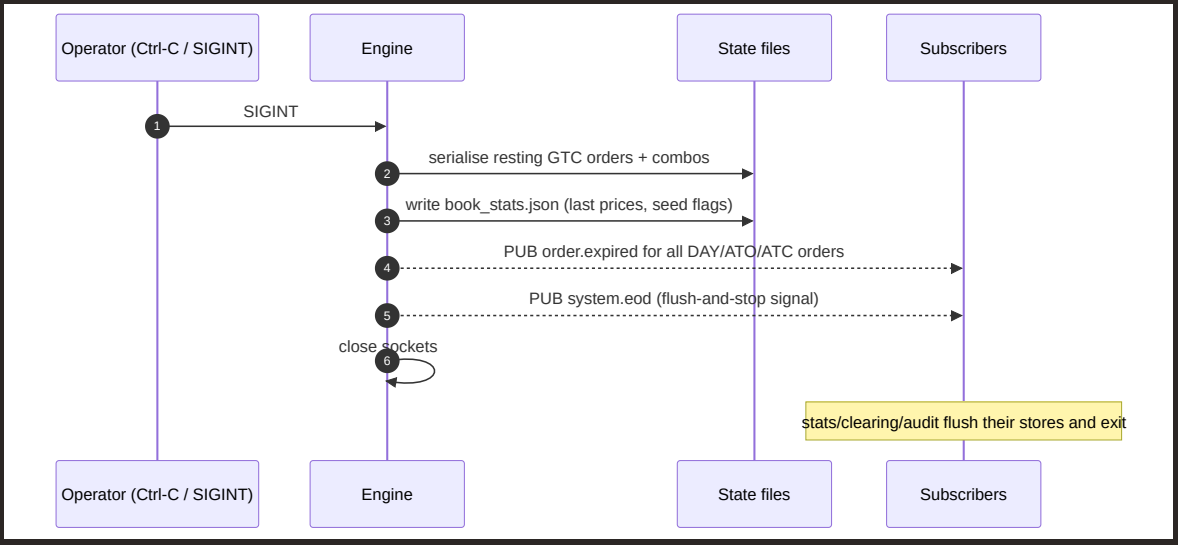
The engine holds the authoritative order book **in memory**. Durability is provided by two independent tiers: engine **state files** written at shutdown and reloaded at startup, and subscriber **data stores** written continuously during the session.

File / store	Written by	Cadence	Survives restart?	Purpose
`gtc_orders.json`	engine	clean shutdown	reloaded at startup	resting GTC orders (price-time priority preserved)
`gtc_combos.json`	engine	clean shutdown	reloaded at startup	resting GTC combo parents + child links
`book_stats.json`	engine	clean shutdown	reloaded at startup	last buy/sell price per symbol; reseeds collar & circuit-breaker references at the next open; each breaker's reference window (`cb_window`), taken back by a same-day restart
`stats.db` (SQLite)	pm-stats	per trade · snapshot every 15 min · EOD · every index.update tick	accumulates	OHLCV, intraday snapshots, trade log, index level snapshots & daily OHLC
`clearing.db` (SQLite)	pm-clearing	per trade · connect/disconnect · EOD	accumulates	positions, VWAP, realised/unrealised P&L

File / store	Written by	Cadence	Survives restart?	Purpose
`audit.log`	pm-audit	continuous (rotating)	accumulates	full chronological event trail
`indexes/<ID>_history.jsonl`	pm-index	on structural/corporate-action event only	accumulates	structural/audit trail (`INIT`, `CORP_ACTION`, `ADD_CONSTITUENT`, `DELIST`) — level/EOD history lives in `stats.db` instead
`indexes/<ID>_state.json`	pm-index	on EOD / corporate action	rewritten	divisor, last prices, and intraday OHLC checkpoint for restart

(Full field-level detail is in the Operator's Guide's Persistence chapter.)

Shutdown is an ordered sequence, not a hard stop — this is what makes GTC recovery and clean consumer flush possible:



At startup the engine reloads `gtc_orders.json` / `gtc_combos.json`, restoring each order **with its original timestamp** so queue priority is unchanged, and reseeds each symbol's collar and circuit-breaker reference from `book_stats.json` — on the same business day the breaker takes back its whole reference window (the trades its reference averages), so a restart halts where an uninterrupted engine would. A market maker's seeded combo restored live is not seeded again.

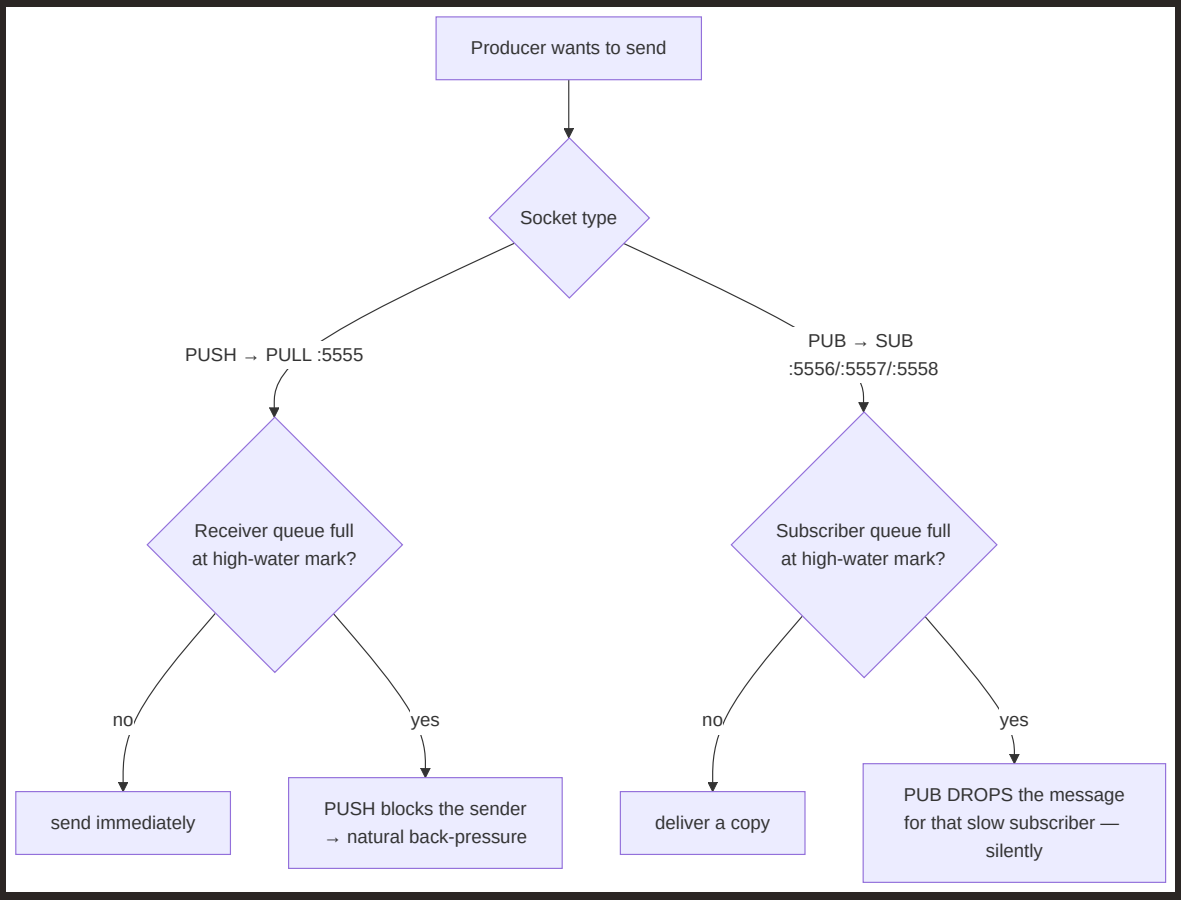
Idempotency and gap detection guard the accumulating stores: trade events are inserted with `INSERT OR IGNORE` on trade id, so a replayed or duplicated event is harmless; the drop-copy feed carries a monotonic sequence number and CALF carries a per-`(channel,symbol)` sequence so consumers can detect and replay gaps.

What is *not* durable. Messages in flight in ZMQ queues are lost on a hard kill (only a clean SIGINT runs the sequence above). DAY orders are intentionally dropped at shutdown. Trade IDs are a per-session monotonic counter that **resets to 1** on restart (`models/trade.py`), so trade-id uniqueness is *within a session only* — merging logs across sessions requires a session prefix. A `SIGKILL` skips GTC serialisation entirely, losing the resting book.

2.12 Failure Modes & Fault Tolerance

The system's resilience is dominated by two facts: there is **one** engine (a single point of failure by design), and the internal bus uses **ZeroMQ default flow control**, which behaves very differently for the two socket patterns.

2.12.1 Back-pressure: PUSH blocks, PUB drops



- **Order entry (PUSH/PULL, :5555):** if the engine falls behind, a gateway's PUSH **blocks** until space frees up. This is benign back-pressure — order submitters slow down; nothing is lost.
- **Event fan-out (PUB/SUB, :5556/:5557/:5558):** if any subscriber can't drain fast enough, ZeroMQ **discards** messages for *that* subscriber once its high-water mark is reached. The loss is **silent** — there is no error, no gap signal from ZMQ itself.

`messaging/bus.py` creates sockets with **default** options — no explicit `SNDHWM`/`RCVHWM`, `LINGER`, or `CONFLATE` are set. So the effective HWM is ZeroMQ's default (~1000 messages per socket).

Silent loss to internal stateful consumers — recommended change

This default-drop behaviour was **inherited, not deliberately chosen**. It is acceptable for viewers (which only ever need the *latest* book) but is a real correctness risk for the **stateful internal consumers that have no replay path** — `pm-clearing` (P&L), `pm-stats` (OHLCV), and `pm-audit` (the compliance log). Under sustained back-pressure these can miss `trade.executed` events and silently diverge, with no gap to detect.

Recommended, in order:

1. **Make it explicit and observable.** Set generous `RCVHWM` on the internal SUB consumers and monitor ZeroMQ's drop counters; a silent gap should become a visible alarm.
2. **Protect the stateful consumers.** For `pm-clearing`/`pm-audit`, prefer a larger queue plus a slow-consumer alarm, or consume via the sequenced feeds (drop-copy / RALF) that support replay, rather than the best-effort `:5556` broadcast.
3. **Conflate the viewers.** `pm-viewer`/`pm-board` only need current state — `zmq.CONFLATE` (keep only the newest message) removes them as a back-pressure source entirely.

Reliable, replay-backed delivery already exists on the **external** edges (CALF `replay_window_sec`, RALF `replay_retention_sec`, drop-copy's 10k in-memory buffer); the gap is on the internal broadcast bus.

2.12.2 Other failure modes

Trigger	Effect	Mitigation / current behaviour
Slow / stalled subscriber	PUB drops its messages at HWM (silent)	see recommendation above; external feeds replay
Engine crash (hard)	whole exchange down; in-flight + DAY orders + resting book lost (no GTC flush)	single-writer SPOF by design; no HA. Clean SIGINT preserves GTC
Engine restart (clean)	DAY orders expire; GTC/combos restored; trade-id counter resets to 1; subscribers reconnect automatically	ordered shutdown + startup reload
Subscriber crash	isolated — engine and other consumers unaffected	reconnects and resumes; may have a gap (replay if supported)
Gateway disconnect	per-gateway `disconnect_behaviour` (`CANCEL_QUOTES_ONLY` default / `CANCEL_ALL` / `LEAVE_ALL`)	configured per gateway
Wall-clock regression	none — priority tie-break stays strictly increasing	`now_ns()` monotonic guarantee
Drop-copy port (:5557) in use	engine logs a warning and runs **without** the drop-copy feed	deliberate: a port conflict must not stop matching
Long auction uncross	blocks all message processing until it finishes	acceptable for teaching; single-thread trade-off

2.13 Performance Optimizations

A sequence of pure-Python changes (no C extensions, no Cython, no multiprocessing) raised single-thread throughput by roughly **2.8×**. This section explains each technique and *why* it works.

Throughput is strongly CPU-architecture dependent

The headline "after" figure of ~**160,000 orders/second** is measured on **ARM (Apple Silicon)**. The identical code on a typical **Intel x86** desktop reaches roughly ~**80,000 orders/second** — about half. The "before" baseline (~57,000/s) and the latency table below are ARM figures. Treat all absolute numbers as *illustrative of the relative gains*, not as a portable benchmark; see Why ARM and Intel differ so much.

2.13.1 How to read the numbers

Every number below was measured with the engine running in a single thread, processing orders through the full hot path (deserialize → validate → match → build messages → publish). "µs" means microseconds (one millionth of a second).

2.13.2 __slots__ on hot-path classes

What it does: By default, Python objects store their attributes in a hidden dictionary (`__dict__`). When you add `__slots__ = ('x', 'y')` to a class, Python stores attributes in a fixed-size C array instead.

Why it's faster:

- Attribute access goes from a hash-table lookup (~60–80 ns) to a direct offset lookup (~30 ns) — roughly 2× faster per access.
- Each instance uses ~40% less memory (no per-object dict allocation), which means less work for the garbage collector.

Where we applied it:

- `Order` (the most common object — one per incoming request)
- `Trade` (one per fill)
- `_HeapEntry` (internal wrapper — thousands live on the book at once)
- `OrderBook` (only a few instances, but accessed on every single order)

Before/after for a dataclass:

```
# Before
@dataclass
class Order:
    id: str
    symbol: str
    ...

# After — just add slots=True
@dataclass(slots=True)
class Order:
    id: str
    symbol: str
    ...
```

For classes that aren't dataclasses (like `_HeapEntry`), you define it manually:

```
class _HeapEntry:
    __slots__ = ('key', 'order', 'valid')

    def __init__(self, key, order, valid=True):
        self.key   = key
        self.order = order
        self.valid = valid
```

Drawbacks: You can no longer add arbitrary attributes at runtime (e.g. `order.debug_tag = "test"` will raise `AttributeError`). Multiple inheritance becomes tricky — all parent classes must also declare `__slots__` or you lose the benefit. Adding a new field requires updating the `__slots__` tuple, which is easy to forget.

2.13.3 Fast enum lookup dictionaries

What it does: Replaces `Side("BUY")` with a pre-built dictionary lookup `_SIDE_MAP["BUY"]`.

Why it's faster: Python's `Enum(value)` constructor iterates through *all* members comparing each string (~600–800 ns). A dictionary lookup is ~50 ns. With 5 enums per order, this saves ~3 µs per deserialization call.

```
# Build once at module load time
_SIDE_MAP = {v.value: v for v in Side}      # {"BUY": Side.BUY, "SELL": Side.SELL}
_TYPE_MAP = {v.value: v for v in OrderType}

# Then in from_dict():
side = _SIDE_MAP[d["side"]]   # ~50 ns instead of Side("BUY") at ~700 ns
```

Drawbacks: If you add a new enum member, you must remember that the lookup dict was built at import time — it won't automatically include the new member unless you rebuild it. Also, an invalid value like `_SIDE_MAP["INVALID"]` raises a `KeyError` instead of the more descriptive `ValueError` that `Side("INVALID")` would give, making debugging slightly harder.

2.13.4 Single timestamp call per order (cached nanosecond clock)

What it does: Reads the clock once at the top of the hot path and passes the result (`now`, an integer nanosecond value) down to every function that needs a timestamp.

Why it's faster: reading the clock is a system call — it crosses from Python into the OS kernel and back (~300–500 ns). A single aggressive order that triggers stops could read it 4–6 times. Caching it means we pay the cost exactly once.

The ordering-critical timestamp is `now_ns()` (`models/clock.py`) — a wrapper over `time.time_ns()` that guarantees a **strictly increasing** integer nanosecond value (it never returns a duplicate or a value that went backwards, even if the wall clock does). `now_ns()` takes a lock to enforce that guarantee across threads. The engine's hot path calls `now_ns()` itself rather than the raw `time.time_ns()` — even though time priority is actually driven by engine arrival sequence, not by this timestamp — because a wall-clock regression (an NTP step, for example) must never make event timestamps go backwards. The lock's cost is small and is a deliberate trade-off, not one the hot path bypasses.

```
def _handle_new_order(self, payload):
    now = now_ns()                      # one clock read, int nanoseconds
    ...
    trades, events = book.process(order, now=now)  # passed through

# Inside the order book:
def _apply_fill(self, aggressor, passive, qty, price, trades, events, now):
    trade = Trade.create(..., now=now)   # reuses cached timestamp
```

Drawbacks: All fills and stop triggers within the same order share the exact same timestamp, even if matching takes a few microseconds — you lose sub-microsecond sequencing *within* a single order (across orders, monotonicity still holds). It also pollutes function signatures — every internal method carries an extra `now` parameter, making the code slightly harder to read and test.

2.13.5 Monotonic integer trade IDs (replacing `uuid4()`)

What it does: Generates trade IDs as sequential integers (`1`, `2`, `3`, …) instead of random UUIDs.

Why it's faster: `uuid.uuid4()` reads from `/dev/urandom` (a kernel call) and then formats 128 random bits into a hyphenated string — total cost ~1.5 µs. `next(counter)` is a pure-Python operation costing ~30 ns.

```
import itertools

_trade_counter = itertools.count(1)

@classmethod
def create(cls, ...):
    return cls(id=str(next(_trade_counter)), ...)
```

Trade IDs only need to be unique within a single engine session, not globally, so a monotonic counter is safe here.

Drawbacks: IDs are no longer globally unique — if you restart the engine, counter resets to 1 and old trade IDs may collide with new ones. This makes it unsafe to merge trade logs from multiple sessions without adding a session prefix. Also, sequential IDs leak information (e.g. competitors can estimate your trade volume by observing the ID gap between two of their fills).

2.13.6 `orjson` instead of stdlib `json`

What it does: Replaces `json.dumps(payload).encode()` with `orjson.dumps(payload)`.

Why it's faster: `orjson` is a C-extension JSON serializer that:

- Encodes directly to `bytes` (no intermediate `str` → `.encode()` step)
- Uses SIMD instructions for string escaping
- Is ~9× faster than stdlib json for typical 10-key dictionaries

Serializer	Cost per call
`json.dumps().encode()`	~2,100 ns

Serializer	Cost per call
`orjson.dumps()`	~230 ns

With 2–4 messages published per order, this alone saved ~4–7 µs on aggressive orders.

```
try:
    import orjson
    def _dumps(obj): return orjson.dumps(obj)
except ImportError:
    import json
    def _dumps(obj): return json.dumps(obj).encode()
```

The fallback ensures the code works in environments where `orjson` isn't installed — it just runs slower.

Drawbacks: Adds a third-party dependency (`orjson`) that must be installed and kept updated. `orjson` is stricter than stdlib `json` — it rejects `NaN`, `Infinity`, and non-string dict keys that stdlib silently accepts. If your payloads ever contain these edge cases, you'll get a `TypeError` at runtime. The library is also platform-specific (compiled C/Rust), so it may not be available on all architectures (e.g. some Alpine Docker images).

2.13.7 Eliminate redundant serialization (`to_dict()` removal)

What it does: Instead of calling `order.to_dict()` to build a full 25-key dictionary and then passing it to a message function, we build only the keys the message actually needs, directly from object attributes.

Why it's faster: `to_dict()` unconditionally copies 25 fields and calls `.value` on every enum — cost ~3–4 µs. The ack message only needs 6 of those fields. Building a minimal dict inline costs ~250 ns.

```
# Before — wasteful
self.pub_sock.send_multipart(
    make_fill_msg(evt.gateway_id, evt.id, ..., order=evt.to_dict())
)

# After — only the fields the consumer needs
self.pub_sock.send_multipart([
    fill_topic_bytes,
    _dumps({
        "order_id": evt.id,
        "fill_qty": evt.quantity - evt.remaining_qty,
        "status":   evt.status.value,
        ...
    }),
])
```

Drawbacks: The message schema is now implicitly defined at each call site instead of in one canonical `to_dict()` method. If a downstream consumer adds a field requirement, you need to find and update every inline dict that builds that message type. Forgetting one is a subtle bug. It also makes writing integration tests harder — you can't just mock `to_dict()` and check its output.

2.13.8 Inlined message construction

What it does: Bypasses helper functions like `make_ack_msg()` and builds the two-frame ZMQ message (`[topic_bytes, payload_bytes]`) directly at the call site.

Why it's faster: Each helper function allocates a dict, conditionally merges fields via `.update()`, then calls `encode()` which does *another* function call. That's 3 function calls + 2 dict allocations. Inlining collapses all of that into a single dict literal + one `orjson.dumps()`:

Approach	Cost
`make_ack_msg(...)`	~950 ns
Inlined with `_dumps()`	~450 ns

Drawbacks: Code duplication — the message format is now repeated at each call site instead of living in one helper. If the message protocol changes (e.g. adding a `"version"` field to all messages), you must update every inline construction point. The engine code also becomes longer and denser, making code reviews harder. For low-frequency messages (rejects, cancels), the savings are negligible and the readability cost isn't justified.

2.13.9 Pre-cached topic bytes

What it does: ZMQ topic strings like `"order.ack.GW01"` are the same for every message sent to a given gateway. We encode them once and store the `bytes` in a dictionary, avoiding repeated `f"order.ack.{gw}".encode()` calls.

Why it's faster: `f-string + .encode()` costs ~100 ns per call. With 3–4 messages per order, that's ~300–400 ns wasted on creating the same bytes. A dict lookup is ~50 ns total.

```
# Populate on first use
self._topic_cache[gw] = f"order.ack.{gw}".encode()

# Hot path
topic = self._topic_cache[gw]  # ~50 ns instead of ~100 ns
```

Drawbacks: The cache grows unboundedly — if gateways connect and disconnect frequently with unique IDs, the dict accumulates stale entries (a minor memory leak). In practice this is a non-issue since gateway IDs are static, but in a general-purpose system you'd want eviction logic. It also adds a layer of indirection that can confuse readers unfamiliar with the pattern.

2.13.10 Local variable caching in tight loops

What it does: Before entering the matching loop, we copy frequently-accessed object attributes into local variables.

Why it's faster: In CPython, local variable access (`LOAD_FAST` bytecode) is a direct array-index operation (~30 ns). Attribute access (`LOAD_ATTR`) involves a descriptor lookup even with `__slots__` (~50–70 ns). In a loop that runs 5–50 iterations, this adds up.

```
def _sweep(self, aggressor, opposite_heap, ...):
    # Cache once before the loop
    _side       = aggressor.side         # LOAD_FAST inside loop
    _smp_action = aggressor.smp_action
    _peek       = self._peek             # avoid self.__dict__ lookup per iteration

    while aggressor.remaining_qty > 0 and opposite_heap:
        best = _peek(opposite_heap)      # local call, not self._peek(...)
        if _side == Side.BUY and best.price > price_limit:
            break
        ...
```

Drawbacks: Makes the code less obvious — a reader seeing `_peek(heap)` has to scroll up to understand it's actually `self._peek`. If the cached attribute is mutable and gets reassigned on `self` during the loop (unlikely here, but possible in other contexts), the local copy becomes stale and introduces bugs. Debugging is also harder because inspecting `self._peek` in a debugger won't show the value actually used inside the loop.

2.13.11 `__new__` for fast deserialization

What it does: Uses `object.__new__(cls)` + direct slot writes instead of calling the dataclass-generated `__init__` with 26 keyword arguments.

Why it's faster: CPython's function-call machinery needs to parse and bind each keyword argument — with 26 kwargs this alone costs ~400 ns. `__new__` creates a bare instance in ~80 ns, and then each `o.field = value` is a simple `STORE_ATTR` (~30 ns with slots).

```
@classmethod
def from_dict(cls, d: dict) -> "Order":
    o = object.__new__(cls)
    o.id         = d["id"]
    o.symbol     = d["symbol"]
    o.side       = _SIDE_MAP[d["side"]]
    ...
    return o
```

Drawbacks: Completely bypasses `__init__`, so any validation, default value assignment, or `__post_init__` logic in the dataclass is skipped. If you later add a new field with a default value to the dataclass, `from_dict` won't automatically pick it up — you'll get an `AttributeError` on first access. This pattern also breaks the implicit contract that dataclass instances are always fully initialized, which can confuse static analysis tools like mypy.

2.13.12 Summary

#	Technique	Savings per order	Applies to
1	`__slots__`	~1.5–2 µs	All hot-path objects
2	Enum lookup dicts	~3 µs	`Order.from_dict()`
3	Cached timestamp	~1–2 µs	Fills + stop triggers
4	Monotonic trade IDs	~1.5 µs	Every fill
5	`orjson`	~4–7 µs	Every published message
6	No redundant `to_dict()`	~3 µs	Ack + fill messages
7	Inlined messages	~500 ns	Ack + fill + trade
8	Pre-cached topic bytes	~300 ns	All messages
9	Local var caching	~200–800 ns	Sweep loop
10	`__new__` deserialization	~400 ns	`Order.from_dict()`

Result:

Metric	Before	After	Change
Throughput	57,000 TPS	160,000 TPS	**+180%**
Median latency	23.5 µs	8.5 µs	**−64%**
P90 latency	25.7 µs	10.0 µs	**−61%**

All of these improvements are **pure Python** — no C extensions, no multi-threading, no unsafe hacks. The key insight is that in a tight loop, small costs (100 ns here, 200 ns there) compound rapidly. At ~160,000 orders/second (ARM), each order has a budget of only **6.25 µs** — every nanosecond matters. On Intel x86 the budget is roughly double (~12.5 µs at ~80,000/s), because the per-order work is the same but each interpreter step is slower (below).

2.13.13 Why ARM and Intel differ so much

The **same code** runs ~2× faster on Apple Silicon (ARM) than on a typical Intel x86 desktop. This is a property of the *workload* — a CPython bytecode interpreter doing pointer-chasing and small-object allocation — meeting two very different microarchitectures. The dominant hypotheses (in rough order of expected impact; confirming the split requires profiling, e.g. `perf stat` on each host):

Factor	Why it favours Apple Silicon
Interpreter dispatch	CPython's eval loop is a giant computed-goto over bytecodes — a torrent of hard-to-predict indirect branches. Apple's very wide decode and large branch-prediction resources swallow this better than mainstream x86 desktop parts.
Memory latency & cache	Refcounting and dict/attribute lookups are latency-bound pointer chases. Apple Silicon's on-package unified memory and large L1/L2 (and huge SLC) cut the average miss cost that this workload pays constantly.
Single-thread IPC / clocks	The engine is single-threaded, so only 1-core performance matters. High-IPC M-series cores at competitive clocks beat many Intel desktop cores on this specific scalar, branchy, allocation-heavy code.
Refcount write traffic	Every `INCREF`/`DECREF` is a small write; Apple's store buffering and cache bandwidth absorb the churn with less stall.
Native library codegen	`orjson` (Rust) and the Python build itself may be compiled with better-tuned codegen / SIMD paths on `arm64` than the x86 wheels in use.

Two caveats. First, "Intel" and "ARM" are not monolithic: a recent high-clock Xeon/Core will close much of the gap versus an older desktop part, and the numbers here compare the specific machines used. Second, the *relative* speedups of the ten techniques above are stable across both architectures — only the absolute TPS moves. If you publish throughput figures, always state the CPU.

2.14 Configuration Choices That Impact Performance

Several config values are not just feature toggles — they directly set the system's work rate, memory footprint, and back-pressure thresholds. The most important is the market-data throttle.

Setting	Where	Default	Performance effect
`snapshot_interval_sec`	engine config	`0.5`	**Dominant market-data lever.** Caps how often `book.{SYM}` / `depth.{SYM}` are published per symbol. Lower = fresher books but more messages, more serialisation, and more downstream CPU across every subscriber and bridge. The floor is the engine poll tick.
Engine poll timeout	`engine/main.py` const	`200 ms`	Upper bound on idle→snapshot latency and on how often circuit-breaker resume timers are checked. Also the effective minimum snapshot cadence.
`market_data_gateway.depth_levels`	CALF	`10`	Bytes per `DEPTH`/`SNAP` line. Combined with a `SYM=*` wildcard subscription, multiplies per-client bandwidth by the symbol count.
`*.max_client_queue`	all gateways	`10000`	Per-client outbound backlog before `SLOW_CLIENT`/drop; also the per-client memory ceiling.
`market_data_gateway.replay_window_sec` · `post_trade_gateway.replay_retention_sec`	CALF / RALF	`30 · 86400`	Replay reach vs replay-buffer memory.
`*.heartbeat_interval_sec` · `*.idle_timeout_sec`	all gateways	varies	Idle-traffic volume vs dead-peer detection latency.
`*.max_commands_per_second` · `*.max_messages_per_second` · `api rate_limit`	ALF / BALF / API	`100 · 100 · 10`	Ingress throttles that protect the single-threaded engine from a runaway client.

Setting	Where	Default	Performance effect
ZeroMQ high-water marks	`messaging/bus.py`	**unset (ZMQ default 1000)**	Governs when PUB silently drops to slow subscribers and when PUSH blocks — see Failure modes. Not currently tunable via config.

Rule of thumb: to reduce load, **raise `snapshot_interval_sec`** first (it scales across every consumer at once); to reduce latency for a specific external consumer, tune that gateway's own knobs rather than the global throttle.

2.15 Tick And Time Representation

- Internal prices are stored as integer ticks in core engine/model logic.
- Internal timestamps are stored as integer nanoseconds (`time_ns`).
- Conversion happens at boundaries only:
  - Inbound user/config prices (decimal) -> ticks.
  - Outbound UI/messages/reporting values -> decimal prices.
  - Outbound display timestamps use seconds derived from ns (`ns / 1e9`).
- Matching, queue priority, auction price selection, and stop logic all operate on integer values to avoid float drift.

Monotonic ordering guarantee. Ordering-critical timestamps come from `now_ns()` (`models/clock.py`), which wraps `time.time_ns()` and enforces a **strictly increasing** sequence: if the raw clock returns a value less than or equal to the last one (a duplicate, or a backwards wall-clock step), `now_ns()` returns `last + 1` instead. This matters because the timestamp is the **tie-breaker in price-time priority** — two orders at the same price must never compare equal, or heap ordering (and therefore queue fairness) becomes undefined. The guarantee is lock-protected for every caller, including the engine's single-threaded order loop: the lock's cost is small and is a deliberate trade-off, not one the hot path bypasses, because a wall-clock regression (an NTP step, for example) must never make event timestamps go backwards — see Performance Optimizations.

CHAPTER 3

Guided Tour: Architecture, Data Structures and Code

Learning objectives

After reading this page you will understand:

- How a limit order book works end-to-end, from order arrival to trade publication
- Why the engine uses heaps, lazy deletion, and index maps for speed and correctness
- How session states and auctions change matching behavior across the trading day
- How advanced order types (FOK/IOC/STOP/ICEBERG/OCO/COMBO) are represented and executed
- How the multi-process ZeroMQ architecture decouples matching, gateway, clearing, stats, and audit
- What optimizations materially improve throughput and latency in the current Python implementation
- Which production-grade capabilities are intentionally out of scope and why

  Audience: A developer new to both the codebase and financial systems. This document builds understanding from first principles — starting with the simplest possible trading concept and expanding layer by layer until the full system makes sense. Every concept is immediately connected to the actual code that implements it.

  Note on types. The engine stores **integer tick prices** (e.g. `$150.30` is `15030` for a `$0.01` tick) and **integer nanosecond timestamps** from `now_ns()` (`models/clock.py`). This walkthrough uses those types. A few early illustrative snippets show `float` prices or `time.time()` purely for readability where the exact type is not the point; such cases are called out inline. For the reference treatment of ticks, the monotonic clock, and conversions, see Architecture → Tick and Time Representation.

  This document vs. the Architecture reference. This is the **code-level walkthrough**. For the design reference — data flows, deployment/trust boundaries, persistence & recovery, failure modes, and the performance/config knobs — see Architecture. Where they overlap, treat 01 as the concise reference and this page as the narrative.

3.1 Quick-Reference Glossary

Terms you will encounter throughout this document, defined up front:

Term	Meaning
Resting order	An order sitting in the book, waiting for a counterparty to fill it
Aggressive order	An incoming order that immediately sweeps against resting orders
Passive order	The resting order on the other side of a fill; sets the price
Spread	The gap between the best bid and best ask prices
TIF	Time-In-Force — how long an order stays live
DAY	Order expires at end of the current trading session

Term	Meaning
GTC	Good-Till-Cancelled — order persists across sessions until explicitly cancelled
ATO	At-The-Open — order valid only during the opening auction
ATC	At-The-Close — order valid only during the closing auction
FOK	Fill-Or-Kill — fill the entire quantity immediately or reject entirely
IOC	Immediate-Or-Cancel — fill what you can immediately, cancel the rest
VWAP	Volume-Weighted Average Price — average price weighted by trade size
OHLCV	Open, High, Low, Close, Volume — standard market data for a time period
SMP	Self-Match Prevention — detecting when your own orders would trade with each other
OCO	One-Cancels-Other — two linked orders; filling one automatically cancels the other
P&L	Profit and Loss
Tick	The minimum price increment for an instrument (e.g. $0.01 for US equities)
Aggressor side	Which side (BUY or SELL) was the incoming aggressive order in a fill
Leg risk	The risk that one part of a multi-leg order fills while another part does not, leaving an unintended position
GC	Garbage Collection — Python's automatic memory reclamation; frequent allocation of large objects increases GC pressure (time spent collecting)
BST	Binary Search Tree — a tree data structure where each node has at most two children and values are stored in sorted order for O(log n) search

3.2 Table of Contents

1. Quick-Reference Glossary
2. What Is an Exchange? The One-Sentence Version
3. The Limit Order: The Atom of Trading
4. The Order Book: A Sorted List of Promises
5. The First Match: How Two Orders Become a Trade
6. Data Structures: Why Heaps?
7. The Order Model: Every Field Explained
8. Beyond Limit Orders: The Full Order Type Taxonomy
9. The Trading Day: Auctions and Continuous Matching
10. Complex Orders: Combos and OCO
11. Market-Maker Quotes: Two Orders, One Obligation
12. The Message Bus: Why ZeroMQ?
13. The Full Process Architecture
14. Optimisations: Speed, Memory, and Latency
15. Persistence: Surviving a Restart
16. Clearing and Settlement: Following the Money
17. What Is Missing: The Gap to Production

3.3 How to Use This Document

Read sections 1–5 in order — they build on each other and the later sections will not make sense without them. After Section 5, most sections are self-contained and can be read in any order depending on what you want to understand.

Recommended reading sequence for the codebase itself:

1. `models/order.py` — the atom. Every other file uses `Order`.
2. `engine/order_book.py` — how atoms are organised and matched.
3. `engine/main.py` — how the order book is orchestrated across symbols and sessions.
4. `models/message.py` — what the engine publishes and what format it uses.

5. `alf_console/main.py` — how a user interacts with the engine.
6. `clearing/main.py` — how fills become P&L.

The rest of the system (`stats`, `audit`, `ticker`, `board`, `scheduler`, `ai_trader`) are consumers of events produced by the above. Read them once you understand the core.

3.4 1. What Is an Exchange? The One-Sentence Version

An exchange is a system that takes **buy orders** and **sell orders** from many participants, figures out which buyers and sellers agree on a price, and executes trades between them.

That is the entire job. Everything else — auctions, stop orders, circuit breakers, clearing — is elaboration on that core.

EduMatcher implements this job in Python. It is not production-grade (we will be explicit about what is missing), but it models every core concept faithfully.

3.5 2. The Limit Order: The Atom of Trading

A **limit order** is a participant saying: *"I want to buy X shares of AAPL, but I will not pay more than $150.00."* Or the reverse: *"I want to sell X shares but I will not accept less than $151.00."*

The key word is **limit** — the participant places a maximum (for a buy) or minimum (for a sell) on the price they will accept. This is not the same as saying "buy at $150.00" — it means "buy at $150.00 *or better*."

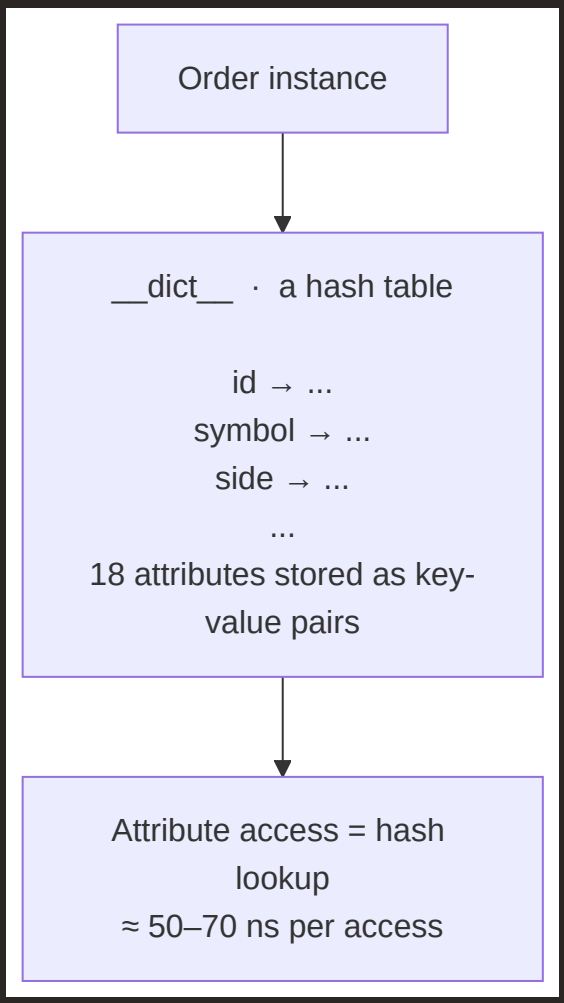
In EduMatcher, this translates directly to the `Order` dataclass. The field names and types below are the **original**, pre-tick-migration ones — simplest to teach the matching logic with — not the current source; Section 6 explains the real current names (`price_ticks`, `ts_ns`, etc.) once the full field list is in view:

```
# models/order.py (original, pre-migration names — see Section 6)
@dataclass(slots=True)
class Order:
    id:           str          # globally unique identifier
    symbol:       str          # which instrument: "AAPL", "MSFT", etc.
    side:         Side         # BUY or SELL
    order_type:   OrderType    # LIMIT, MARKET, STOP, etc.
    quantity:     int          # total shares requested
    remaining_qty: int         # shares not yet filled (starts equal to quantity)
    price:        Optional[float]  # the limit price — None for market orders
    gateway_id:   str          # which participant submitted this
    timestamp:    float        # when it was submitted (not the queue position — see arrival_seq)
    status:       OrderStatus  # NEW → PARTIAL → FILLED (or CANCELLED, REJECTED, EXPIRED)
```

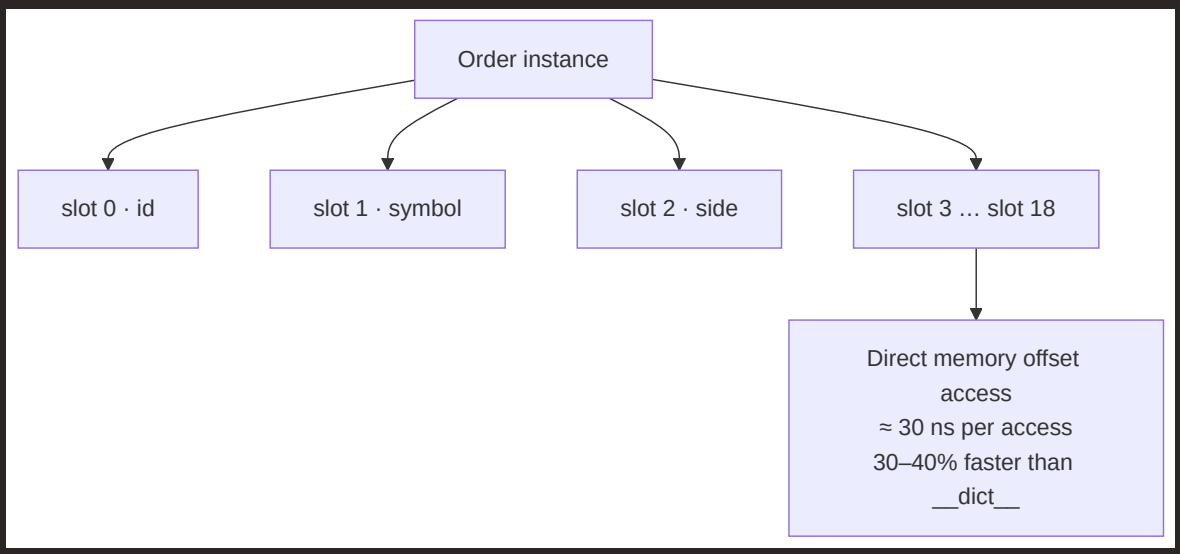
The `remaining_qty` field is crucial — it is how we track partial fills. If you order 100 shares and 30 have traded, `remaining_qty` is 70. The order stays in the book until either `remaining_qty` reaches zero (fully filled) or the order is cancelled.

3.5.1 Why `__slots__`?

Notice `@dataclass(slots=True)`. Without `__slots__`, every Python object stores its attributes in a dictionary:



With `__slots__`, Python replaces the dictionary with a fixed C array:



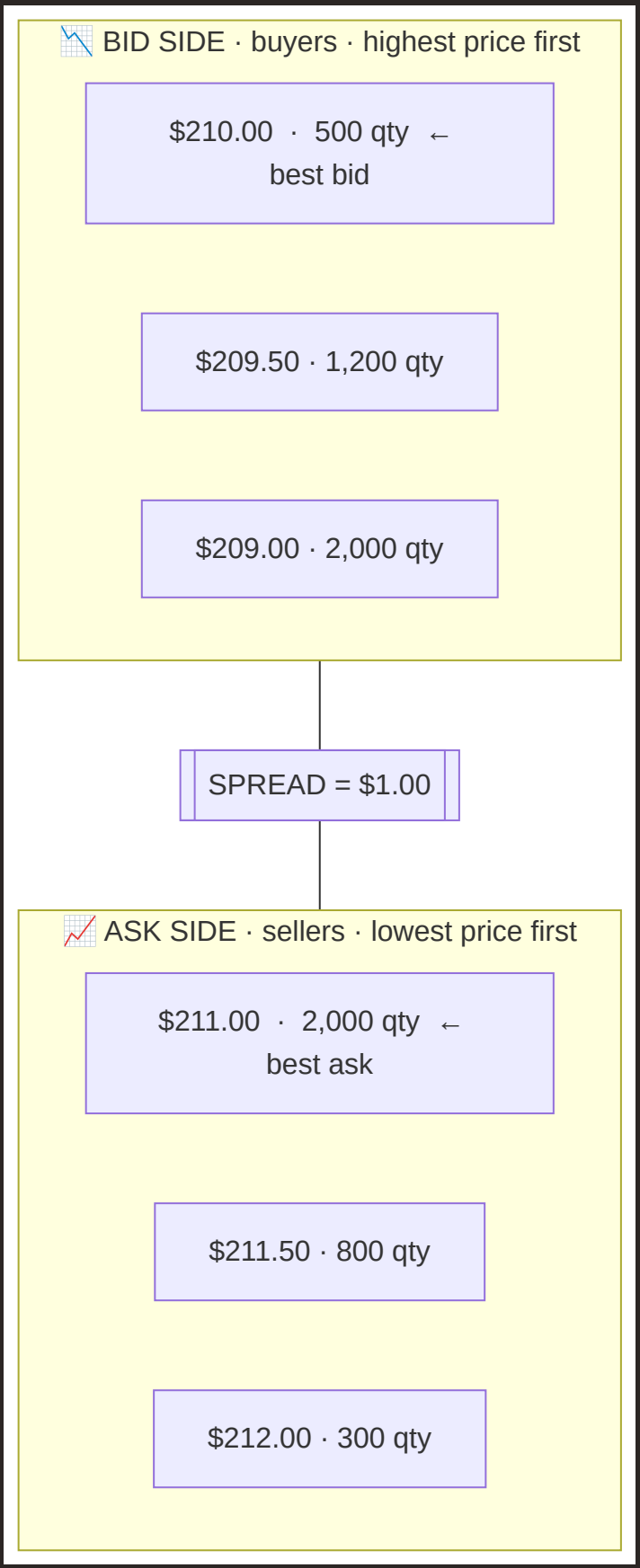
Each attribute access saves roughly 20-40 nanoseconds. With thousands of fills per second and 5+ attribute accesses per fill, this compounds to 1-2 microseconds saved per order — about 15% of the total per-order budget.

Memory savings are also significant: no per-object `__dict__` means each `Order` uses roughly 40% less memory. With thousands of resting orders in the book, this reduces both memory consumption and garbage collection pressure.

3.6 3. The Order Book: A Sorted List of Promises

The **order book** is the core data structure of an exchange. It holds all resting (unfilled) orders, organised by price level, waiting for a counterparty.

Conceptually, it looks like this for AAPL:

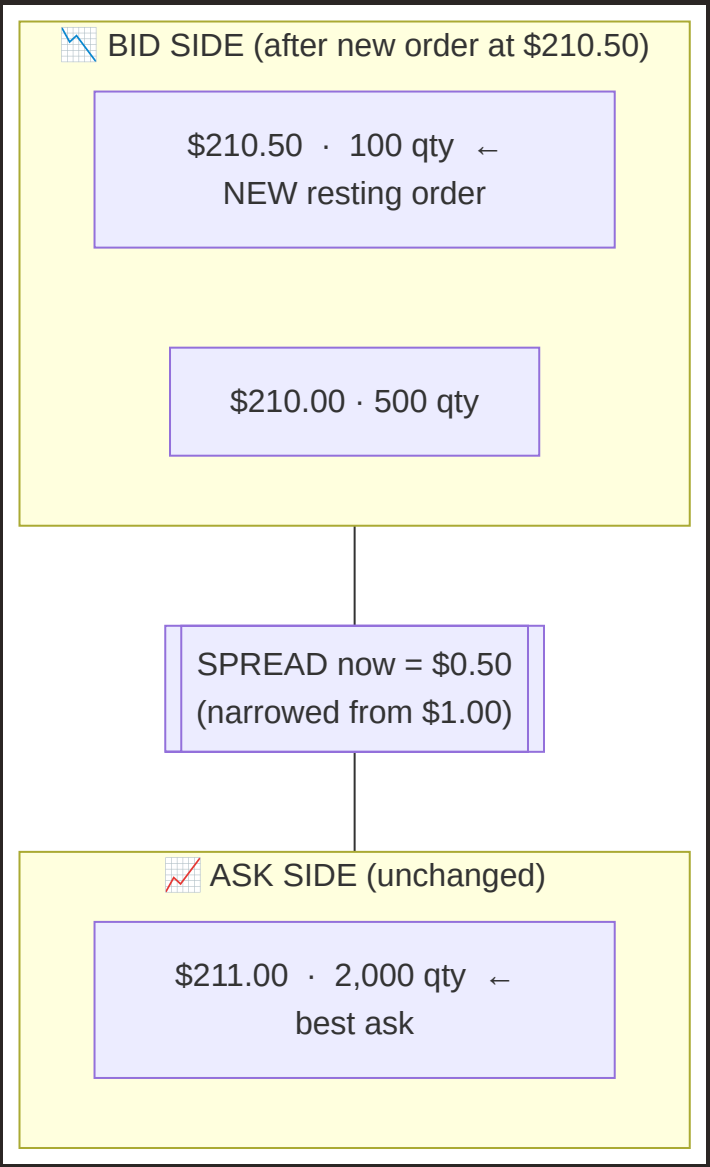


The **spread** is the gap between the best bid ($210.00) and the best ask ($211.00). In this example, the cheapest you can buy immediately is $211.00. The most you can sell immediately is $210.00.

When does a match occur? Two orders can trade when their prices *cross* — the buyer's limit price is at or above the seller's limit price. In the diagram above, if a new buy order arrived at $211.00 or higher, it would match against the resting

sell at $211.00. The book would be crossed and a trade would happen.

When does a match NOT occur? If a new buy order arrives at $210.50, it cannot fill immediately — the cheapest available seller wants $211.00, which is more than the buyer is willing to pay. The new buy order would rest in the book at $210.50, narrowing the spread. The book would now look like:



This accumulation of unfilled limit orders is what creates depth in the book.

In EduMatcher, this is `OrderBook`:

```
# engine/order_book.py
class OrderBook:
    __slots__ = (
        "symbol",
        "_bids",       # max-heap of resting buy orders
        "_asks",       # min-heap of resting sell orders
        "_buy_stops",  # min-heap of buy stop orders
        "_sell_stops", # max-heap of sell stop orders
        "_trailing_stops",  # list of trailing stop orders
        "_bid_qty",    # dict[price → total qty at that price level]
        "_ask_qty",    # dict[price → total qty at that price level]
        "_order_index",  # dict[order_id → Order] — for O(1) cancel lookup
        "_entry_index",  # dict[order_id → _HeapEntry] — for lazy deletion
        "_orders_by_gateway",  # dict[gateway_id → ordered order_ids] — for per-gateway cancel
        "last_trade_price",  # most recent fill price — used by stop trigger checks
        "last_trade_qty",    # quantity of the most recent fill
        "last_buy_price",    # most recent price where the aggressor was a buyer
        "last_sell_price",   # most recent price where the aggressor was a seller
        "recent_trades",     # deque[Trade] maxlen=20 — rolling window of last 20 fills,
                             # published in book snapshots so the viewer can show a
                             # live tape of recent transactions for this symbol
        "daily_qty",         # cumulative session traded quantity (reset only on restart)
        "daily_value_ticks", # cumulative session turnover, in integer ticks to avoid float drift
        "daily_trades",      # cumulative session trade count
        "_has_stops",        # True if the book has any resting stop order — skips the
                             # stop-check call on every fill when no stops are present
        "_arrival_seq",      # monotonic arrival sequence — queue priority is keyed on
                             # this, not the client-supplied timestamp
    )
```

Notice `__slots__` here too — `OrderBook` is accessed on every incoming order. With 21 attributes, every slot optimisation multiplies across every order processed.

There is one `OrderBook` instance per symbol. The engine holds them in a dict:

```
# engine/main.py
self.books: dict[str, OrderBook] = {}  # symbol → OrderBook

def _book(self, symbol: str) -> OrderBook:
    if symbol not in self.books:
        self.books[symbol] = OrderBook(symbol)  # create on first order
    return self.books[symbol]
```

3.7 4. The First Match: How Two Orders Become a Trade

Here is the simplest possible matching scenario:

Scenario: The book has a resting sell order: 100 shares of AAPL at $211.00. A new buy order arrives: 50 shares of AAPL at $215.00 (willing to pay up to $215).

The buy price ($215.00) is higher than the ask price ($211.00). These orders **cross** — the buyer is willing to pay more than the seller requires. A trade happens.

In EduMatcher, this flows through `OrderBook.process()`:

```
def process(self, order: Order, *, match: bool = True, now: int | None = None):
    ...
    elif order.order_type == OrderType.LIMIT:
        self._match_limit(order, trades, events, now)
    ...
```

`_match_limit` calls `_sweep`:

```
def _match_limit(self, order, trades, events, now):
    opposite = self._asks if order.side == Side.BUY else self._bids
    self._sweep(order, opposite, price_limit=order.price, trades=trades,
                events=events, now=now)
    if order.remaining_qty > 0:
        self._rest(order)  # unfilled remainder rests on the book
```

`_sweep` is the core matching loop:

```
def _sweep(self, aggressor, opposite_heap, price_limit, trades, events, now):
    # Cache frequently-accessed attributes as locals for speed.
    # (Section 13 shows the full set of cached locals including _smp_action
    # and _gw_id for self-match prevention; this version is simplified.)
    _side = aggressor.side
    _peek = self._peek
    _apply_fill = self._apply_fill

    while aggressor.remaining_qty > 0 and opposite_heap:
        best = _peek(opposite_heap)    # O(1) — best resting order
        if best is None:
            break

        # Price check: stop if the best available price is too expensive
        if price_limit is not None:
            if _side == Side.BUY and best.price > price_limit:
                break  # cheapest ask is above our limit — stop
            if _side == Side.SELL and best.price < price_limit:
                break  # best bid is below our limit — stop

        fill_qty = min(aggressor.remaining_qty, best.remaining_qty)
        fill_price = best.price
        _apply_fill(aggressor, best, fill_qty, fill_price, trades, events, now)
```

The fill price is **always the resting order's price**, not the aggressor's price. This is a fundamental exchange rule: the passive order (sitting in the book) sets the price. Our buyer offered up to $215 but paid $211 — the resting seller's price. This is beneficial to the aggressive side (buyer pays less than maximum) and is how all continuous limit order markets work.

`_apply_fill` records the trade:

```
def _apply_fill(self, aggressor, passive, fill_qty, fill_price, trades, events, now):
    if aggressor.side == Side.BUY:
        buy_order, sell_order = aggressor, passive
    else:
        buy_order, sell_order = passive, aggressor

    trade = Trade.create(
        symbol=self.symbol,
        buy_order_id=buy_order.id,
        sell_order_id=sell_order.id,
        buy_gateway_id=buy_order.gateway_id,
        sell_gateway_id=sell_order.gateway_id,
        price=fill_price,
        quantity=fill_qty,
        aggressor_side=aggressor.side.value,  # "BUY" or "SELL" — for clearing/stats
        now=now,  # reuse pre-computed timestamp — no extra syscall
    )
    trades.append(trade)
    self.last_trade_price = fill_price

    # Update quantities
    aggressor.remaining_qty -= fill_qty
    passive.remaining_qty -= fill_qty

    # Update statuses
    aggressor.status = FILLED if aggressor.remaining_qty == 0 else PARTIAL
    passive.status   = FILLED if passive.remaining_qty == 0 else PARTIAL
    if passive.status == FILLED:
        self._entry_index[passive.id].valid = False  # lazy deletion
    ...
```

The result: a `Trade` object is produced and added to the `trades` list. The engine will publish this to all subscribers on the message bus. The passive order's heap entry is marked `valid = False` so `_peek` will skip it next time (lazy deletion — explained fully in Section 5).

3.7.1 What Happens to `trades` and `events` After `process()` Returns?

`process()` returns two lists: `trades` (completed fills) and `events` (order state changes). Back in `engine/main.py`, after `book.process()` returns, the engine publishes both:

```
def _handle_new_order(self, payload: dict) -> None:
    ...
    trades, events = book.process(order, match=do_match, now=now)
    self._publish_events(events, book)   # sends fill/cancel acks to gateways
    self._publish_trades(trades)          # sends trade.executed to all subscribers
    self._mark_dirty(symbol)             # schedules a book snapshot
```

`_publish_events()` iterates the `events` list and sends a private fill notification to each affected gateway — the buyer's gateway and the seller's gateway each receive their own message. These messages are private (topic includes the gateway ID) so only the relevant participant sees their own fill.

`_publish_trades()` broadcasts the trade to ALL subscribers on the PUB socket with topic `trade.executed`. The clearing process, stats recorder, and audit log all receive it.

3.7.2 The Partial Fill Case

In the scenario above, the buy order was for 50 shares against a sell order for 100 shares. After the fill:

- **Aggressive order (buy 50)**: `remaining_qty = 0` → status = `FILLED`. It is consumed entirely.
- **Passive order (sell 100)**: `remaining_qty = 50` → status = `PARTIAL`. It remains in the book, still resting at $211.00, waiting for the next buyer.

The next buy order that matches $211.00 would fill against those remaining 50 shares.

3.8 5. Data Structures: Why Heaps?

3.8.1 The Problem

The matching engine's inner loop asks one question thousands of times per second: *"What is the best resting order right now?"*

For bids: best = highest price. For asks: best = lowest price. The structure must support:

1. **O(1) peek at the best order** — called on every incoming aggressive order
2. **O(log n) insert** — every new resting order
3. **O(1) cancel** — without scanning the book

A sorted list satisfies (1) but insert/delete are O(n). A balanced BST (Binary Search Tree — a tree where each node is larger than all nodes in its left subtree and smaller than all in its right, giving O(log n) search) satisfies all three but has significant constant-factor overhead in Python. The EduMatcher solution: **a heap with lazy deletion and a flat index**.

3.8.2 The Heap

Python's `heapq` module implements a **min-heap**: a list where `list[0]` is always the smallest element. Insert and pop are O(log n); reading `list[0]` is O(1).

For asks (we want cheapest first) this is natural. For bids (we want most expensive first) we negate the price:

```
# For asks: price goes in directly — lowest ask stays at heap[0]
key = (order.price, order.arrival_seq)

# For bids: negate price so highest bid sorts first
key = (-order.price, order.arrival_seq)
```

Using a tuple `(price, arrival_seq)` as the key encodes **price-time priority** for free: Python compares tuples element-by-element, so same-price orders are broken by arrival order automatically. `arrival_seq` is a counter the book increments each time it admits an order — deliberately *not* the order's own `timestamp`, which the client supplies and could back-date to jump the queue (see Section 6).

3.8.3 The `_HeapEntry` Wrapper

`Order` objects cannot be compared directly (should a BUY be "less than" a SELL?). The `_HeapEntry` wrapper carries the sort key separately:

```
class _HeapEntry:
    __slots__ = ("key", "order", "valid")

    def __init__(self, key, order, valid=True):
        self.key   = key    # (±price, arrival_seq) — the sort key
        self.order = order  # the Order object — never compared
        self.valid = valid  # False when cancelled or filled

    def __lt__(self, other):
        return self.key < other.key  # heapq only calls __lt__
```

`__slots__` here is critical — `_HeapEntry` is the most frequently allocated object in the system, one per resting order. Without slots, each instance carries a `__dict__` with three keys, using ~200 bytes. With slots: ~72 bytes and ~30% faster attribute access. In a book with 5,000 resting orders this saves ~640KB of memory and reduces GC (Garbage Collection — Python's automatic memory reclamation) pressure significantly.

3.8.4 Lazy Deletion

Python's `heapq` has no `remove()` operation. Finding and removing an element from the middle of a heap would require an O(n) scan plus O(log n) re-heapification — O(n) total. For a high-frequency system, this is unacceptable.

The solution: when an order is cancelled or filled, **do not remove its entry from the heap**. Just set `entry.valid = False`:

```
def cancel_order(self, order_id: str) -> Optional[Order]:
    order = self._order_index.get(order_id)
    if order is None:
        return None
    order.status = OrderStatus.CANCELLED
    entry = self._entry_index.get(order_id)
    if entry:
        entry.valid = False    # O(1) — the entry stays in the heap
        self._deduct_qty_index(entry.order, ...)
    return order
```

The `_peek` method handles lazy cleanup:

```
def _peek(self, heap: list[_HeapEntry]) -> Optional[Order]:
    while heap:
        entry = heap[0]
        if not entry.valid:
            heapq.heappop(heap)  # discard — O(log n), paid once
            continue
        o = entry.order
        if o.status in (FILLED, CANCELLED, REJECTED, EXPIRED):
            heapq.heappop(heap)
            entry.valid = False
            continue
        return o  # valid resting order found
    return None
```

The cost of discarding an invalid entry (one `heappop`) is O(log n), but it is paid only once per entry. In a busy book where orders are frequently cancelled before they fill, this approach replaces O(n) remove operations with O(log n) cleanup spread across many `_peek` calls.

What "amortised O(1)" means for `_peek`: In the *worst* case, a single `_peek` call could pop many stale entries before finding a valid one — O(k log n) where k is the number of stale entries. But each stale entry is paid for exactly once, by the single `_peek` call that removes it. Spread across all `_peek` calls over time, the average cost per call is O(1). This is what "amortised O(1)" means: not O(1) every time, but O(1) on average over many calls.

3.8.5 The Dual-Index Pattern

The book maintains two flat dictionaries in parallel with the heaps:

```
self._order_index:  dict[str, Order]      # order_id → Order
self._entry_index:  dict[str, _HeapEntry] # order_id → heap entry
```

These answer different questions:

- `_order_index`: "Given an order ID, give me the Order object." Used for cancel and amend operations that arrive from the outside with just an order ID.
- `_entry_index`: "Given an order ID, give me its heap entry so I can mark it invalid." Used by the lazy deletion pattern.

Without `_order_index`, cancelling an order requires scanning the heap — O(n). With it, cancel is two O(1) dict lookups plus setting a boolean.

Concrete example — cancelling order `"abc-123"`:

```
# A cancel message arrives with only the order_id — no symbol, no price.
# Without the indexes, we would need to:
#   1. Scan all 5000 entries in _bids to find it — O(n)
# With the dual index:
#   1. _order_index["abc-123"] → the Order object — O(1)
#   2. _entry_index["abc-123"] → the _HeapEntry → set .valid = False — O(1)
#   3. _bid_qty[order.price] -= order.remaining_qty  — O(1)
#      if _bid_qty[order.price] == 0: del _bid_qty[order.price]  — O(1)
```

That last line is important: when the last resting order at a price level is cancelled or filled, the price level entry is deleted from `_bid_qty` (or `_ask_qty`). This keeps the dict compact and ensures best bid/ask and the depth snapshot only see price levels that actually have live orders at them.

3.8.6 The Price-Level Quantity Index

```
self._bid_qty: dict[int, int]  # price_ticks → total qty at that bid level
self._ask_qty: dict[int, int]  # price_ticks → total qty at that ask level
```

This index is maintained in sync with the heap and answers: *"How much total quantity is available at price X?"*

It makes depth statistics and best bid/ask cheap:

Depth metrics — computing how much volume sits within 1% of mid price doesn't require iterating all resting orders, just the price-level dict.

3.8.7 Summary: Why This Combination?

Operation	Complexity	Mechanism
Peek best order	O(1) amortised	heap[0] + lazy skip
Insert new resting order	O(log n)	heappush
Cancel order	O(1)	dict lookup + valid=False
Amend (price change)	O(log n)	invalidate + new heappush
FOK availability check	O(n log n)	sorted walk of live heap
Depth metrics	O(P)	price-level dict
Full book snapshot	O(n)	iterate heap

Where n = resting orders, P = distinct price levels (P « n in practice).

3.9 6. The Order Model: Every Field Explained

> **Migration note:** The field names and types shown below (and used in the code examples throughout Sections 2-5 and 13 of this chapter, for simplicity) reflect the **original** codebase before the tick migration. In the current codebase the names changed too, to make the integer tick/nanosecond representation explicit: `price` → `price_ticks`, `stop_price` → `stop_price_ticks`, `trail_offset` → `trail_offset_ticks`, and `timestamp` → `ts_ns`; all four are now `Optional[int]` (or plain `int` for `ts_ns`) instead of `float`. The semantics are unchanged. See `EduMatcher_Tick_Migration_Plan_v6.md` for the full type changes.

```
@dataclass(slots=True)
class Order:
    # --- Identity ---
    id:           str       # UUID — globally unique, set at creation
    symbol:       str       # "AAPL", "MSFT" — which instrument

    # --- Order specification ---
    side:         Side      # BUY or SELL
    order_type:   OrderType # LIMIT, MARKET, STOP, STOP_LIMIT, FOK, IOC, ICEBERG, TRAILING_STOP
    tif:          TIF       # Time-In-Force: DAY, GTC, ATO, ATC
    quantity:     int       # original total requested
    remaining_qty: int      # how much is still unfilled
    price:        Optional[float]  # limit price; None for pure market orders
    tick_decimals: int      # decimal scale of every price/qty tick field, at the
                             # time this order was created — stored, not looked up
                             # live, so a restored order keeps the scale it was
                             # written with even if the symbol's config changes later

    # --- Stop order fields ---
    stop_price:   Optional[float]  # STOP/STOP_LIMIT/TRAILING_STOP trigger
    trail_offset: Optional[float]  # TRAILING_STOP: distance from market

    # --- Iceberg fields ---
    visible_qty:   Optional[int]   # fixed peak size (e.g. 100 of 1000)
    displayed_qty: Optional[int]   # current visible slice (replenished after fill)

    # --- Execution context ---
    gateway_id:   str       # which participant submitted this
    timestamp:    float     # client-supplied submission time (display/audit only)
    status:       OrderStatus  # NEW → PARTIAL → FILLED or CANCELLED/REJECTED/EXPIRED
    arrival_seq:  int       # engine-assigned monotonic arrival sequence — this, not
                             # the client-supplied timestamp, is what price-time
                             # priority is actually keyed on (a back-dated payload
                             # timestamp cannot jump the queue); 0 = unassigned
    cancel_reason: Optional[str]  # why the *exchange* cancelled this order (SMP, an
                             # unfillable MARKET/IOC, etc.) — set by OrderBook, not
                             # the client
    is_seed:      bool      # True when created by config-driven startup seeding
                             # (market_maker_quotes/combos) rather than a live
                             # gateway submission — purely observational
    client_tag:   Optional[str]  # client-supplied correlation tag, echoed back on
                             # every lifecycle event; ignored when absent

    # --- Self-match prevention ---
    smp_action:   SmpAction # NONE, CANCEL_AGGRESSOR, CANCEL_RESTING, CANCEL_BOTH

    # --- Complex order group membership ---
    oco_group_id:    Optional[str]  # if part of a One-Cancels-Other pair
    combo_parent_id: Optional[str]  # if a leg of a combo order
    leg_index:       Optional[int]  # position in combo legs list (0-based)

    # --- Quote origin (added in the feature plan) ---
    origin:   OrderOrigin   # ORDER (default) or QUOTE (created from a market-maker quote)
    quote_id: Optional[str] # the QuoteEntry ID this leg belongs to; None for regular orders
```

The four SMP actions represent different business policies for what to do when your own order is about to trade against your other order:

- **NONE** — no prevention. Allow self-trading. This is dangerous (creates artificial volume, potentially manipulative) but sometimes used in testing or by systems that manage their own SMP at a higher level.
- **CANCEL_AGGRESSOR** — cancel the incoming order. The resting order stays. Use this when you want to protect your standing orders: "I am quoting at $150 and I do not want my own algorithmic orders sweeping my own quotes." The

resting order continues to be available to other participants.

- **CANCEL_RESTING** — cancel the resting order instead, and continue sweeping the next order in the queue. Use this when your aggressive order is more important than your resting order: "I am sending a new best price and it should supersede my old price, not bounce off it." The sweep loop calls `continue` to look at the next resting order, which may belong to a different participant.
- **CANCEL_BOTH** — cancel both. Use when you want to clean up: "Neither of these orders should exist simultaneously."

`CANCEL_AGGRESSOR` is the most common default in practice.

3.9.1 Time-In-Force (TIF): How Long Does the Order Live?

DAY — expires at market close. The most common default. On engine shutdown, all DAY orders receive an `order.expired` notification.

GTC (Good-Till-Cancelled) — persists until explicitly cancelled or filled. Survives engine restarts (saved to `data/gtc_orders.json` on shutdown, restored on startup). Institutional participants often use GTC to maintain standing limit orders.

ATO (At-The-Open) — valid only during the opening auction. The engine rejects ATO orders submitted outside `OPENING_AUCTION` state:

```
# engine/main.py
if self._sessions_enabled and order.tif == TIF.ATO and
   self._session_state != SessionState.OPENING_AUCTION:
    # reject with reason "ATO orders only accepted during opening auction"
```

ATC (At-The-Close) — same pattern, valid only during `CLOSING_AUCTION`.

3.9.2 The Enum Design Pattern

All enums in EduMatcher inherit from both `str` and `Enum`:

```
class Side(str, Enum):
    BUY  = "BUY"
    SELL = "SELL"
```

`Side.BUY == "BUY"` evaluates to `True`. This means enum members can be compared directly to JSON strings, eliminating conversion in hot paths. When you receive a JSON message with `"side": "BUY"`, you can compare directly without calling `Side("BUY")` first.

For deserialization, fast lookup dicts replace the slow `Enum(value)` constructor:

```
# Built once at module load — O(1) lookup vs O(n) scan inside Enum()
_SIDE_MAP:   dict[str, Side]        = {v.value: v for v in Side}
_TYPE_MAP:   dict[str, OrderType]   = {v.value: v for v in OrderType}
_STATUS_MAP: dict[str, OrderStatus] = {v.value: v for v in OrderStatus}
```

`Enum("BUY")` iterates all members doing string comparisons — ~600-800ns per call. Dict lookup is ~50ns. With 5 enum constructions per `Order.from_dict()` call (called on every incoming order), this saves ~3-4µs per order — roughly 15-20% of the deserialization budget.

3.9.3 The `from_dict` Bypass

The `Order.from_dict()` method bypasses the `dataclass __init__` entirely:

```
@classmethod
def from_dict(cls, d: dict[str, Any]) -> "Order":
    # PERF: bypass __init__ — saves ~400ns from argument dispatch overhead
    o = object.__new__(cls)
    o.id        = d["id"]
    o.symbol    = d["symbol"]
    o.side      = _SIDE_MAP[d["side"]]
    o.order_type = _TYPE_MAP[d["order_type"]]
    o.tif       = _TIF_MAP[d["tif"]]
    ...
    return o
```

The normal dataclass `__init__` with 26 keyword arguments spends ~400ns on argument dispatch alone. Writing slot values directly via `__new__` eliminates this.

3.10 7. Beyond Limit Orders: The Full Order Type Taxonomy

3.10.1 Market Order

A market order says "buy/sell at whatever price is available right now." No price limit. In `_match_market`:

```
def _match_market(self, order, trades, events, now):
    opposite = self._asks if order.side == Side.BUY else self._bids
    self._sweep(order, opposite, price_limit=None, ...)  # no price check
    if order.remaining_qty > 0:
        order.status = OrderStatus.CANCELLED  # unfillable remainder: discard
```

Note: market orders **cannot rest**. If there is not enough liquidity to fill the entire quantity, the remainder is discarded (cancelled). This is correct exchange behaviour — a market order without a price limit cannot sit in the book because there is nothing to price it at.

3.10.2 IOC (Immediate-Or-Cancel)

Like a limit order but with a strict rule: fill what you can at the limit price, cancel any unfilled remainder immediately. Never rests in the book.

```
def _match_ioc(self, order, trades, events, now):
    self._sweep(order, opposite, price_limit=order.price, ...)
    if order.remaining_qty > 0:
        order.status = OrderStatus.CANCELLED  # unlike LIMIT: never rests
        events.append(order)
```

IOC is useful for participants who want price certainty but are not willing to wait in the queue.

3.10.3 FOK (Fill-Or-Kill)

The most restrictive type: either the entire quantity fills immediately or the entire order is rejected. No partial fills, no resting.

```
def _match_fok(self, order, trades, events, now):
    opposite = self._asks if order.side == Side.BUY else self._bids
    available = self._fok_available_qty(order, opposite)
    if available < order.quantity:
        order.status = OrderStatus.REJECTED
        events.append(order)
        return  # reject entirely — no fills
    self._sweep(order, opposite, price_limit=order.price, ...)
```

Note the pre-check (`_fok_available_qty` in the real code; the snippet above is simplified). It walks the live resting orders in priority order up to the limit price — counting an iceberg's hidden reserve and excluding liquidity that self-match prevention would skip — *before* any fills are attempted. The `_bid_qty`/`_ask_qty` index is not used here because it holds only visible quantity. This is important for correctness: if the check fails, zero fills have occurred, so there is no partial state to unwind.

3.10.4 Stop Orders

A stop order is a **conditional order**: it becomes active (converts to a market or limit order) when the market price reaches the stop price.

- **Buy stop** — triggers when price **rises** to `stop_price`. Used to buy into upward momentum or to limit losses on a short position.
- **Sell stop** — triggers when price **falls** to `stop_price`. Used to cut losses on a long position ("stop-loss").

Stop orders rest in separate heaps, not the bid/ask book:

```
self._buy_stops:  list[_HeapEntry]  # min-heap keyed by (stop_price, arrival_seq)
self._sell_stops: list[_HeapEntry]  # max-heap keyed by (-stop_price, arrival_seq)
```

After every fill, the engine checks whether any stops should trigger:

```
def _check_stops(self, now: int) -> list[Order]:
    if self.last_trade_price is None:
        return []

    triggered = []

    # Buy stops: fire when last_price >= stop_price
    while self._buy_stops:
        entry = self._buy_stops[0]
        stop_price, _ = entry.key
        if self.last_trade_price < stop_price:
            break  # KEY INSIGHT: heap is sorted — if top hasn't triggered,
                   # none below it will either. O(1) in the common case.
        heapq.heappop(self._buy_stops)
        # Convert stop → market or limit
        stop_order = entry.order
        if stop_order.order_type == OrderType.STOP:
            stop_order.order_type = OrderType.MARKET
            stop_order.price = None
        else:  # STOP_LIMIT
            stop_order.order_type = OrderType.LIMIT
        triggered.append(stop_order)

    # ... similar for sell stops
    return triggered
```

The `break` is the critical optimisation. Because the heap is sorted by stop price, if the top entry has not triggered, none of the entries deeper in the heap can have triggered either. This makes stop checking O(1) in the common case (no stops firing) and O(k log k) when k stops fire. A naive linear scan would be O(s) on every single trade, where s is the total number of pending stops.

3.10.5 STOP vs STOP_LIMIT: A Critical Difference

When a stop order triggers, it converts into a different order type. The choice has a significant risk implication:

- **STOP** → converts to a **MARKET** order. It will fill at whatever price is available. If the market has moved sharply ("gapped"), you might fill at a much worse price than your stop. For example: your sell stop is at $145. The last trade is at $140 (a $5 gap down overnight). Your stop triggers and converts to a market sell — you fill at $140, not $145.
- **STOP_LIMIT** → converts to a **LIMIT** order at the original limit price. It will only fill at that price or better. This protects you from gap fills — but the order may not fill at all if the market has gapped past the limit. Continuing the example: your sell stop_limit at $145 converts to a limit sell at $145. The market is now at $140 — no buyer wants to pay $145. Your order rests unfilled, and your exposure continues.

Neither is strictly better; the choice depends on whether you prioritise execution certainty (STOP) or price certainty (STOP_LIMIT).

3.10.6 Trailing Stop Orders

A trailing stop is a stop whose trigger price **moves with the market**. A sell trailing stop with offset $2 starts at `last_price - $2`. As price rises, the stop rises with it. If price falls, the stop holds and eventually triggers.

```
for order in self._trailing_stops:
    if order.side == Side.SELL:
        # Ratchet up: if market rises, tighten the stop
        candidate = trade_price - order.trail_offset
        if candidate > order.stop_price:
            order.stop_price = candidate  # stop can only move up, never down

        # Trigger: price fell to the stop
        if trade_price <= order.stop_price:
            order.order_type = OrderType.MARKET
            triggered.append(order)
            continue

    still_active.append(order)

self._trailing_stops = still_active  # lazy deletion via list rebuild
```

Trailing stops use a simple list rather than a heap because their stop price changes on every trade. Maintaining heap order would require re-heapifying on every price update — expensive. The list is iterated fully each time, but trailing stop lists are typically short.

3.10.7 Iceberg Orders

An iceberg order hides most of its quantity. It shows only a `visible_qty` (the "tip") in the public book; the hidden reserve is not visible to other participants.

Why use an iceberg? A participant wanting to buy 10,000 shares of a thinly traded stock faces a problem: if they show 10,000 shares as a resting buy, other participants see the large demand and may raise their ask prices. By showing only 100 shares at a time, the buyer avoids revealing their true intention and gets better prices.

Concrete example: An iceberg BUY order arrives with `quantity=1000`, `visible_qty=100`. In the book, only 100 shares are visible. A sell order arrives and fills those 100 shares. The iceberg replenishes: a new 100-share peak is created from the remaining 900 hidden shares. This continues until either all 1,000 shares fill or the order is cancelled. Each replenishment assigns a fresh `arrival_seq`, putting the iceberg at the back of its price level's queue.

```
# Resting: only displayed_qty is shown to the market
def _rest(self, order: Order) -> None:
    qty = (
        order.displayed_qty           # iceberg: show only the peak
        if order.order_type == OrderType.ICEBERG
        else order.remaining_qty      # regular: show full quantity
    )
    ...
    self._bid_qty[order.price] = self._bid_qty.get(order.price, 0) + qty
```

After the visible slice is consumed, it replenishes from the hidden reserve and goes to the **back of the queue** at its price level:

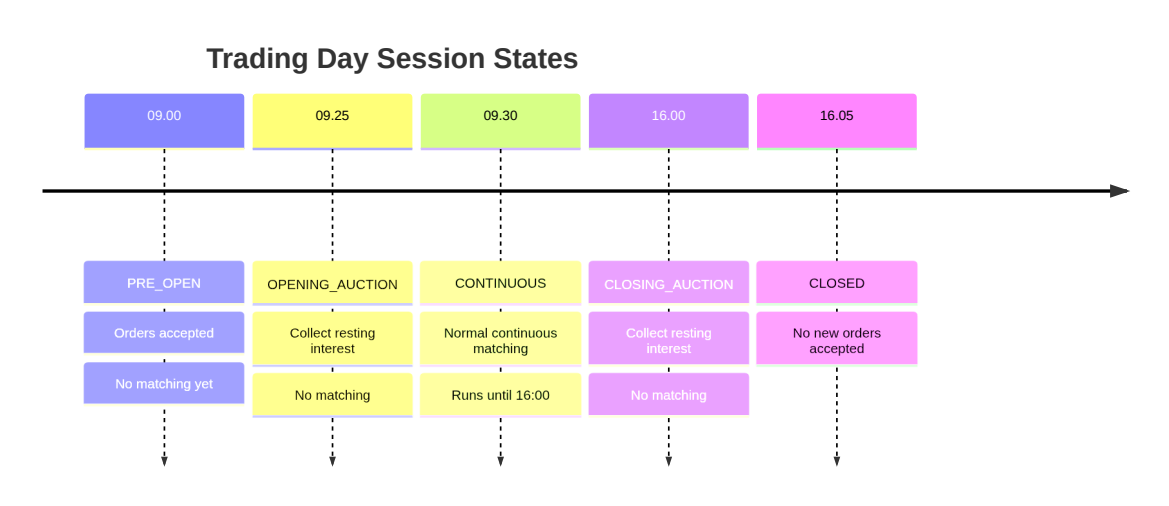
Why back of the queue? Fairness. Other participants who have been patiently waiting at the same price level should not be indefinitely displaced by an iceberg that keeps recycling to the front. Each time the iceberg replenishes, it takes its turn at the back, just like any other new order arriving at that price.

```
# In _apply_fill, after filling a passive iceberg:
if passive.remaining_qty > 0 and passive.displayed_qty == 0:
    new_peak = min(passive.visible_qty, passive.remaining_qty)
    passive.displayed_qty = new_peak
    passive.timestamp = now        # bookkeeping only — see below
    self._reinsert_iceberg(passive)  # invalidate old entry, push new one
```

`_reinsert_iceberg` is the key mechanism, not the timestamp update: it assigns the order a **fresh `arrival_seq`** (Section 6 — this, not the timestamp, is what the heap key is actually built from) and pushes a new heap entry under that later sequence number, correctly losing queue priority to every order already resting at that price.

3.11 8. The Trading Day: Auctions and Continuous Matching

Real exchanges do not simply run continuous matching all day. A typical trading day looks like this:



This is modelled in `models/session.py`:

```
class SessionState(str, Enum):
    PRE_OPEN         = "PRE_OPEN"
    OPENING_AUCTION  = "OPENING_AUCTION"
    CONTINUOUS       = "CONTINUOUS"
    CLOSING_AUCTION  = "CLOSING_AUCTION"
    CLOSED           = "CLOSED"

VALID_TRANSITIONS = {
    SessionState.PRE_OPEN:        {SessionState.OPENING_AUCTION, SessionState.CONTINUOUS},
    SessionState.OPENING_AUCTION: {SessionState.CONTINUOUS},
    SessionState.CONTINUOUS:      {SessionState.CLOSING_AUCTION, SessionState.CLOSED},
    SessionState.CLOSING_AUCTION: {SessionState.CLOSED},
    SessionState.CLOSED:          {SessionState.PRE_OPEN},
}
```

The `VALID_TRANSITIONS` dict enforces that the engine cannot jump from, say, `CLOSED` directly to `CONTINUOUS` — sessions must flow through the defined states.

Why enforce transitions explicitly? Without this guard, a bug in the scheduler (or a test script that sends transitions in the wrong order) could put the engine in an impossible state — for example, jumping from `CLOSED` directly to `CONTINUOUS`, bypassing the opening auction entirely. Real exchange rules require that every trading day begins with an opening auction to establish a fair opening price. Encoding the valid transitions explicitly ensures the engine's session logic can never be violated by external inputs.

3.11.1 The Scheduler: A Separate Process

Transitions are driven by the `pm-scheduler` process — a completely separate Python process that sends `session.transition` messages to the engine over ZMQ:

```
# scheduler/main.py
for hhmm, state in schedule:
    target = _time_today(hhmm)
    wait_secs = (target - datetime.now()).total_seconds()
    time.sleep(wait_secs)
    push_sock.send_multipart(make_session_transition_msg(state))
```

The key design insight: **the engine does not know what time it is**. It receives transitions from outside. This means the scheduler can be replaced with a test harness that fires all transitions immediately (`--now` mode), without changing the engine at all.

3.11.2 No-Matching Mode: Collecting Auction Interest

During auction phases, `OrderBook.process()` is called with `match=False`:

```
def process(self, order, *, match: bool = True, now=None):
    if not match:
        if order.order_type in (OrderType.MARKET, OrderType.FOK, OrderType.IOC):
            order.status = OrderStatus.REJECTED
            return trades, events  # these cannot rest
        self._rest(order)  # LIMIT/ICEBERG rest without matching
        return trades, events
```

Market, FOK, and IOC orders are rejected during auctions — they require immediate execution which is not available. Limit orders simply rest and accumulate.

3.11.3 The Auction Uncross

At the end of an auction, `compute_equilibrium` finds the price that maximises executable volume:

```
# engine/auction.py
def compute_equilibrium(book: "OrderBook") -> AuctionResult:
    # Build cumulative buy and sell quantities at each price level.
    # Not from _bid_qty/_ask_qty: that index holds only an iceberg's
    # visible peak, and the uncross executes hidden quantity too.
    bid_qty, ask_qty = _auction_levels(book)   # one O(n) walk of the heaps
    bid_prices = sorted(bid_qty.keys(), reverse=True)
    ask_prices = sorted(ask_qty.keys())

    # For each candidate price P:
    # buy_qty  = Σ bids at prices >= P  (buyers willing to pay at least P)
    # sell_qty = Σ asks at prices <= P  (sellers willing to accept at most P)
    # exec_qty = min(buy_qty, sell_qty) — how much actually trades
    # Pick P that maximises exec_qty; break ties by minimising surplus
```

For example, imagine this auction book:

```
Price    Cumul. bids (qty willing to pay ≥ P)    Cumul. asks (qty willing to accept ≤ P)    Executable
$209     1,000                                    0                                          0   (all sellers want ≥$210; none will t
$210       700                                  300                                        300   min(700, 300) = 300
$211       500                                  700                                        500   min(500, 700) = 500  ← maximum
$212       200                                  900                                        200
```

Cumulative bids at price P = total quantity of buy orders with a limit price *at or above P* — buyers willing to pay at least P. As P rises, fewer buyers qualify, so the cumulative count falls.

Cumulative asks at price P = total quantity of sell orders with a limit price *at or below P* — sellers willing to accept P or anything higher. As P falls, fewer sellers qualify, so the cumulative count also falls.

Equilibrium price = $211 (maximises volume at 500 lots)

After uncross: 500 trades happen at $211 · 500 lots of buy orders at $211+ are filled · 500 lots of sell orders at $211 and below are filled · Remaining: 200 buy orders still resting (bid surplus)

Tie-breaking: if two candidate prices produce the same executable volume, the algorithm picks whichever has the lower surplus (buy/sell imbalance at that price); if that is tied too, the lowest of the tied prices wins, because candidates are scanned in ascending order and only a strict improvement in either measure replaces the current best (`engine/auction.py::compute_equilibrium`). There is no reference to the previous close here.

After finding the price, `execute_uncross` fills all crossable interest:

```
def execute_uncross(book, eq_price):
    now = now_ns()  # from models/clock.py — single strictly-increasing
                        # nanosecond timestamp for all auction fills
    while True:
        best_bid = book._peek(book._bids)
        best_ask = book._peek(book._asks)
        if best_bid is None or best_ask is None:
            break
        if best_bid.price < eq_price: break  # remaining bids below equilibrium
        if best_ask.price > eq_price: break  # remaining asks above equilibrium
        fill_qty = min(best_bid.remaining_qty, best_ask.remaining_qty)
        book._apply_fill(best_bid, best_ask, fill_qty, eq_price, ...)
```

All auction fills happen at the single equilibrium price, simultaneously. The `now = now_ns()` outside the loop means all trades in one auction uncross share the same timestamp — reflecting their logical simultaneity.

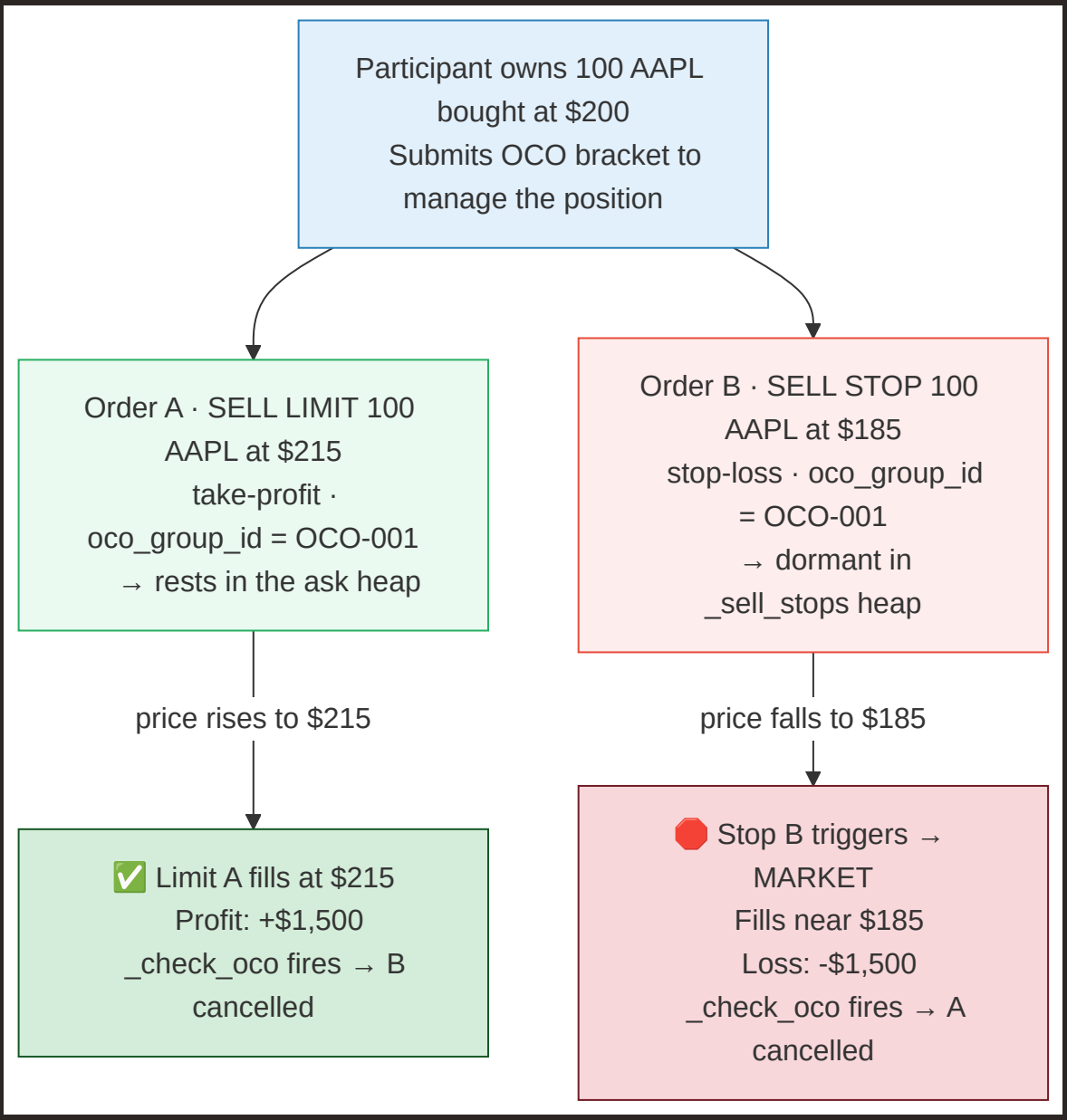
3.12 9. Complex Orders: Combos and OCO

Iceberg orders are covered in Section 7. This section covers the two order types that link multiple orders together at the engine level.

3.12.1 OCO (One-Cancels-Other)

An OCO pair consists of two orders linked together. When either fills — or is otherwise cancelled — the other is automatically cancelled too. A typical use: "I bought AAPL at $200 and I want to automatically take profit at $215, but also automatically cut my loss if price falls to $185." Both legs of an OCO pair must be on the **same symbol** (and share the same quantity and TIF); the engine takes one `symbol` for the whole pair, not one per leg, so a position cannot be hedged OCO-style by an order on a different, correlated symbol.

Concrete OCO lifecycle — take-profit at $215, stop-loss at $185:



Only one of outcome 2a or 2b ever occurs — whichever fills first cancels its sibling. The participant is guaranteed to exit their position at one of the two prices.

```
# engine/main.py
self._oco_groups:   dict[str, list[str]] = {}  # oco_id → [order_id_1, order_id_2]
self._order_to_oco: dict[str, str]       = {}  # order_id → oco_id
```

After either leg reaches a terminal state — not just `FILLED`, but also `CANCELLED` or `REJECTED` — the sibling is cancelled too, so a leg that is cancelled for any other reason (e.g. a risk action) tears down the pair just like a fill would:

```
def _check_oco_after_event(self, order: Order) -> None:
    oco_id = order.oco_group_id
    order_ids = self._oco_groups.get(oco_id)
    sibling_ids = [oid for oid in order_ids if oid != order.id]
    for sibling_id in sibling_ids:
        symbol = self._order_symbol.get(sibling_id)
        book = self.books.get(symbol)
        if book:
            cancelled = book.cancel_order(sibling_id)
            if cancelled:
                self.pub_sock.send_multipart(make_oco_cancelled_msg(...))
    self._oco_groups.pop(oco_id, None)
```

3.12.2 Combo Orders (Multi-Leg)

A combo order ties multiple single-instrument orders together. The implemented semantics is AON (All-Or-None): the combo is considered complete only when all legs fill. If any leg fails, the remaining legs are cancelled.

```
# models/combo.py
@dataclass
class ComboOrder:
    id:              str          # internal UUID
    combo_id:        str          # user-provided label
    gateway_id:      str
    combo_type:      ComboType    # AON
    tif:             TIF
    legs:            list[ComboLeg]
    child_order_ids: list[str]    # populated after child orders are created
    leg_fill_qty:    dict[int, int]  # leg_index → filled quantity
    leg_statuses:    dict[int, str]  # leg_index → OrderStatus value
```

The engine decomposes a combo into individual child orders, one per leg, and posts them to their respective symbol books:

```
def _accept_combo(self, combo: ComboOrder, ...) -> bool:
    self.pub_sock.send_multipart(self._combo_ack(..., True))
    self._combos[combo.id] = combo
    for i, leg in enumerate(combo.legs):
        child = Order.create(symbol=leg.symbol, side=leg.side, ...)
        child.combo_parent_id = combo.id
        child.leg_index = i

        self._order_to_combo[child.id] = combo.id
        book = self._book(leg.symbol)
        trades, events = book.process(child)
        self._publish_dispatch(trades, events, ...)
        if combo.status is ComboStatus.FAILED:
            break
```

The combo is registered before its first leg is posted, so a leg that is rejected or cancelled on entry fails the combo through the same follow-up as any later event, and the legs after it are never posted.

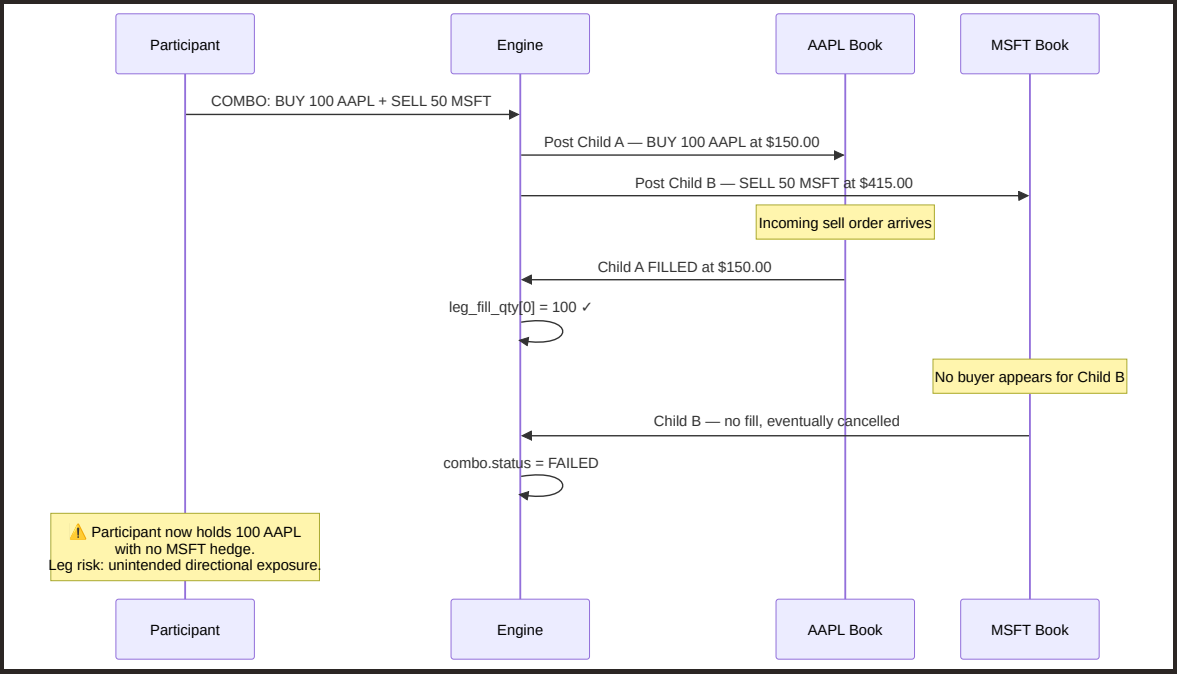
When a child fills, the engine updates the parent combo's status. When all legs fill, the combo is marked `MATCHED`:

```
def _check_combo_after_child_event(self, order: Order) -> None:
    combo_id = self._order_to_combo.get(order.id)
    combo = self._combos.get(combo_id)
    combo.leg_statuses[order.leg_index] = order.status.value
    combo.leg_fill_qty[order.leg_index] = order.quantity - order.remaining_qty

    if combo.is_fully_filled:
        combo.status = ComboStatus.MATCHED
        self.pub_sock.send_multipart(make_combo_status_msg(..., "MATCHED"))
    elif order.status in (CANCELLED, EXPIRED, REJECTED):
        # Cascade cancel: one leg failed, cancel all remaining
        combo.status = ComboStatus.FAILED
        for child_id in combo.child_order_ids:
            if child_id != order.id:
                symbol = self._order_symbol.get(child_id)
                book = self.books.get(symbol)
                if book:
                    book.cancel_order(child_id)
```

Important limitation: this implementation has **leg risk** — the risk that one leg fills while another does not, leaving an unintended open position. Child orders are posted to independent books. If Leg A fills but Leg B never fills, the combo reaches `FAILED` and Leg B is cancelled — but Leg A's fill is a real, committed trade. It cannot be reversed. True atomic combo matching requires a dedicated combo book where both sides must match simultaneously. This is not implemented.

Concrete legging risk scenario:



This is why production combo matching uses a dedicated combo order book where both legs are evaluated together before any fills are committed.

3.13 10. Market-Maker Quotes: Two Orders, One Obligation

A `QUOTE` is how a market maker posts continuous two-sided liquidity: one message carries a bid and an ask together, and replacing it is a single atomic operation rather than two independent cancel/new pairs. Underneath, a quote is nothing more than two ordinary `LIMIT` orders in the book — same heaps, same price-time priority, same matching logic as Section 4. What is different is bookkeeping: the engine needs to answer "which two orders belong to this gateway's current quote on this symbol?" cheaply, and it needs to keep that answer correct through fills, cancels, disconnects, and restarts. This section covers the two indexes that make that possible, and the one cancellation path that has to fall back to a third mechanism when both indexes have already forgotten about a leg.

3.13.1 The Two Orders

`Engine._handle_quote_new` builds a quote's bid and ask exactly like any other order, then tags them:

```
# engine/main.py — _handle_quote_new()
bid = Order.create(
    symbol=symbol, side=Side.BUY, order_type=OrderType.LIMIT,
    quantity=bid_qty, gateway_id=gateway_id, tif=tif,
    price=bid_price, smp_action=smp_action,
)
ask = Order.create(
    symbol=symbol, side=Side.SELL, order_type=OrderType.LIMIT,
    quantity=ask_qty, gateway_id=gateway_id, tif=tif,
    price=ask_price, smp_action=smp_action,
)
bid.origin = OrderOrigin.QUOTE
ask.origin = OrderOrigin.QUOTE
bid.quote_id = quote_id
ask.quote_id = quote_id
```

`origin` and `quote_id` are not `Order.create()` parameters — they are set afterward, because they exist to answer "was this order created by a `QUOTE` message, and which one?", a question ordinary `NEW` orders never need to ask. Both legs are then rested via `book.process(...)` exactly like any other limit order.

3.13.2 Index 1 — `QuoteIndex`: Which Two Orders Make Up *This* Quote?

`models/quote.py`'s `QuoteIndex` is the engine-level index of *active* quotes, one entry per `(gateway_id, symbol)`:

```
# models/quote.py
class QuoteIndex:
    def __init__(self, history_maxlen: int = DEFAULT_QUOTE_HISTORY_MAXLEN) -> None:
        self._index: dict[tuple[str, str], QuoteEntry] = {}
        self._keys_by_gateway: dict[str, dict[tuple[str, str], None]] = {}
        self._keys_by_symbol: dict[str, dict[tuple[str, str], None]] = {}
        self._history: dict[str, deque[QuoteHistoryEntry]] = {}
```

`_index` is the primary map: `(gateway_id, symbol) → QuoteEntry`, where a `QuoteEntry` is little more than `{quote_id, bid_order_id, ask_order_id}`. At most one entry exists per `(gateway_id, symbol)` — a `put()` for a key that already holds an entry silently replaces it, mirroring the fact that a gateway can only ever have one live quote per symbol. `_keys_by_gateway`/`_keys_by_symbol` are secondary indexes over the same key space, so "cancel everything this gateway has quoted" (`cancel_all_for_gateway`, used by gateway disconnect and kill switch) and "cancel every quote on this symbol" (`cancel_all_for_symbol`, used by circuit breakers and symbol halts) are O(k) in the number of that gateway's/symbol's active quotes, not a scan of every quote in the engine. They are insertion-ordered dicts used as sets, so a batch cancel publishes in the order the quotes arrived: a plain `set` of string keys iterates in the per-process hash order, and two runs of the same input would publish differently.

`QuoteIndex` answers exactly one question well: *"what are the two order ids for gateway G's current quote on symbol S?"* It does not know whether those two orders are still resting, still partially filled, or already gone — it is a bookkeeping record of intent, not a live view of the book. That gap is exactly why a second index exists.

3.13.3 Index 2 — `OrderBook._orders_by_gateway`: Which Orders Are *Still Resting*, Per Gateway?

`QuoteIndex` is popped the moment a fill inactivates a quote (see below) — but under the engine's default refresh policy, a *partial* fill deliberately leaves the hit leg's remainder resting, live and tradeable, with no `QuoteIndex` entry pointing at it any more. Something still needs to find that leftover order when the market maker's next quote arrives, without paying for a scan of every resting order in the book to do it. `OrderBook` carries a small secondary index for exactly this:

```
# engine/order_book.py
self._orders_by_gateway: dict[str, dict[str, None]] = {}
```

`gateway_id → order_ids` (an insertion-ordered dict used as a set, for the same reason as `QuoteIndex`'s keys), scoped to this one symbol's book, holding every order — any origin, any type — that is currently resting there. It started life scoped to `origin == OrderOrigin.QUOTE` only, built for exactly the quote-reissue question this section opens with; it was broadened to cover every resting order once an unrelated problem turned out to need the identical shape of index (see "A Second Consumer: Mass-Cancel on Disconnect" below). It is maintained by every funnel that can make an order start or stop resting — insert in `_rest()`, `_add_stop()`, `_add_trailing_stop()`; remove in `_purge_from_indexes()` and the direct `_order_index` pops in `_check_stops()`/`_check_trailing_stops()` — so it follows the same "Dual-Index Pattern" discipline as `_order_index`/`_entry_index` from Section 5:

```
# _rest() — plain and quote-leg LIMIT orders
self._order_index[order.id] = order
self._entry_index[order.id] = entry
self._orders_by_gateway.setdefault(order.gateway_id, {})[order.id] = None
```

```
# _add_stop() / _add_trailing_stop() — STOP, STOP_LIMIT, TRAILING_STOP
# (a separate insertion funnel from _rest(), see Section 7)
self._order_index[order.id] = order
self._orders_by_gateway.setdefault(order.gateway_id, {})[order.id] = None
```

```
# _purge_from_indexes() — the sole point where an order stops resting
# via cancel_order() or _apply_fill()'s full-fill branch
entry = self._entry_index.pop(order.id, None)
if entry is not None:
    entry.valid = False
self._order_index.pop(order.id, None)
self._discard_from_gateway_index(order)  # shared by every removal path below
```

A triggered stop is a fourth removal path, and a subtler one: `_check_stops`/ `_check_trailing_stops` pop a triggered order out

of `_order_index` directly, *before* `_purge_from_indexes` ever runs, because a triggered `STOP` that matches immediately (converts to `MARKET`, never rests again) would otherwise leak in `_orders_by_gateway` forever — nothing else would ever discard it. Both methods call the same `_discard_from_gateway_index` helper `_purge_from_indexes` uses. A triggered `STOP_LIMIT`, by contrast, converts to `LIMIT` and typically rests again immediately afterward through the normal `_rest()` call — so it is discarded and re-added within the same `process()` call, never double-counted and never missing.

`orders_for_gateway(gateway_id)` is the read side — an O(k) lookup (k = that gateway's resting orders in this book) instead of the O(n) `resting_orders()` scan it replaces:

```
def orders_for_gateway(self, gateway_id: str) -> list[Order]:
    order_ids = self._orders_by_gateway.get(gateway_id, ())
    return [self._order_index[oid] for oid in order_ids]
```

`quote_orders_for_gateway(gateway_id)` — the method this section originally motivated — is now a thin origin filter on top of it:

```
def quote_orders_for_gateway(self, gateway_id: str) -> list[Order]:
    return [o for o in self.orders_for_gateway(gateway_id) if o.origin == OrderOrigin.QUOTE]
```

Self-match prevention's `_smp_cancel_resting` funnels through `_purge_from_indexes` too, for the same reason `cancel_order` and `_apply_fill` do: a resting order — quote leg or otherwise — can be the "resting" side SMP cancels when a later order from the same gateway crosses its own still-live order, and that removal has to be visible to this index exactly like any other.

3.13.4 A Second Consumer: Mass-Cancel on Disconnect

`Engine._handle_gateway_disconnect`'s `CANCEL_ALL` behaviour, and the three kill-switch handlers (`_handle_kill_switch`, `..._gateway`, and the per-symbol branch of `..._global`), all need to answer "which of gateway G's orders are resting in this book?" — the same question this index already answered for quote legs, just without the `origin == QUOTE` filter. Before this index covered every origin, those handlers scanned `resting_orders()` — every resting order in every book — and filtered in Python for a matching `gateway_id`. On a real exchange, walking every resting order across every book on every gateway disconnect is not a performance nitpick: with enough symbols and enough resting orders, that sweep runs long enough to block the (deliberately single-threaded, see Section 12) engine thread for a visible stretch, during which no other gateway's orders can be acknowledged, matched, or cancelled either — the kind of latency spike a real venue's risk and monitoring would treat as an incident, not a shrug. `orders_for_gateway` turns that O(total resting orders) scan into O(k) per book (k = that one gateway's resting orders there):

```
# engine/main.py — _handle_gateway_disconnect(), CANCEL_ALL branch
for book in self.books.values():
    for order in list(book.orders_for_gateway(gateway_id)):
        if order.origin != OrderOrigin.QUOTE:
            self._cancel_order_by_id(order.id)
```

Quote-origin orders are still excluded here — not because the index cannot see them, but because `_quote_index.cancel_all_for_gatew` called just above this loop, already retires them through `QuoteIndex`'s own path (quote cancellation publishes `quote.status` messages this loop does not know how to produce).

Two admin handlers were deliberately left alone: `_handle_kill_switch_global` (cancels every gateway, not one) and `_handle_cancel_s` (cancels every gateway on one symbol). Neither is keyed by a single gateway, so `orders_for_gateway` does not help them — a true "cancel everyone in scope" sweep has to touch every resting order in that scope regardless of how it is indexed, and `resting_orders()` remains the right tool for that job.

3.13.5 Why Two Indexes, Not One

`QuoteIndex` and `_orders_by_gateway` look like they overlap, but they answer different questions and go stale at different moments:

	`QuoteIndex`	`_orders_by_gateway`
Lives in	`Engine` (one instance, all symbols)	`OrderBook` (one per symbol)
Answers	"What is gateway G's *current* quote (by `quote_id`) on symbol S?"	"Which of gateway G's orders in *this book* are still resting, of any origin/type?"

	`QuoteIndex`	`_orders_by_gateway`
Entry removed when	A fill inactivates the quote, a replace/cancel supersedes it, a disconnect/kill-switch/halt clears it	The specific order is cancelled or fully filled — regardless of `QuoteIndex`'s state
Survives a partial fill?	No — inactivation policy can remove the entry immediately	Yes — the hit leg's resting remainder stays indexed until it actually stops resting

The second row is the crux: after `INACTIVATE_ON_ANY_FILL` pops `QuoteIndex` on a partial fill, `QuoteIndex` has *forgotten this quote ever existed*, but the hit leg's remainder is still real, resting, matchable liquidity — and `_orders_by_gateway` (via `quote_orders_for_gateway`'s origin filter) is the only structure still tracking it. Without it, finding that leftover on the next quote would mean scanning every resting order in the book.

3.13.6 The Fallback: Cancelling a Leftover Leg `QuoteIndex` No Longer Knows About

`Engine._handle_quote_new`'s replace logic tries `QuoteIndex` first:

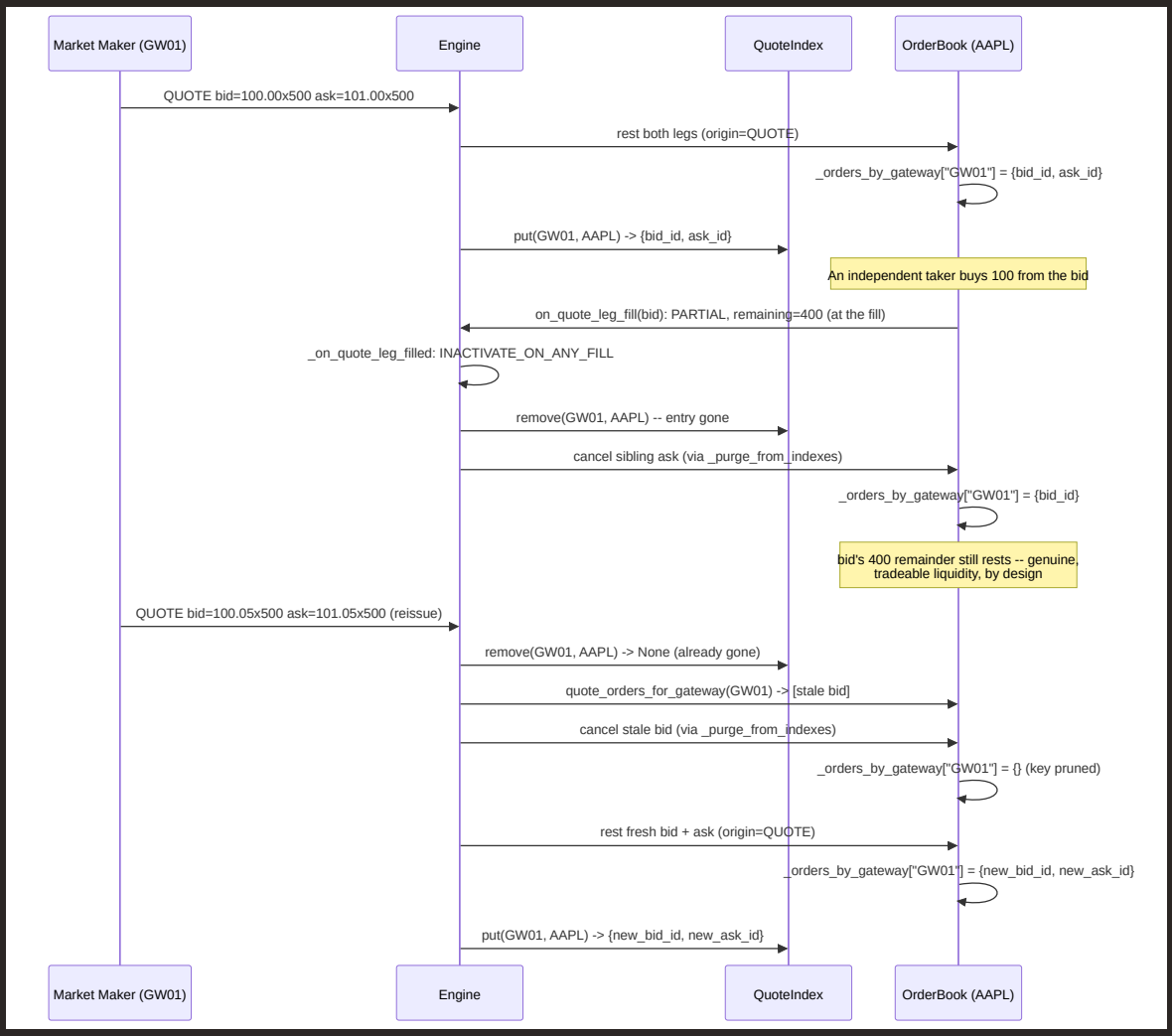
```
# engine/main.py — _handle_quote_new()
previous = self._quote_index.remove(gateway_id, symbol, reason="Replaced by new quote")
if previous:
    self._cancel_quote_entry(previous, reason="Replaced by new quote")
else:
    self._cancel_orphaned_quote_legs(gateway_id, symbol)
```

The `if previous` branch is the common, cheap case: `QuoteIndex` still has an entry, both legs (whichever are still resting) get cancelled through it, and `_orders_by_gateway` is updated as a side effect of each cancel going through `_purge_from_indexes`. The `else` branch is what runs after a partial fill already popped `QuoteIndex`:

```
# engine/main.py
def _cancel_orphaned_quote_legs(self, gateway_id: str, symbol: str) -> int:
    book = self.books.get(symbol)
    if book is None:
        return 0
    cancelled = 0
    for order in list(book.quote_orders_for_gateway(gateway_id)):
        if self._cancel_order_by_id(order.id) is not None:
            cancelled += 1
    return cancelled
```

This is an O(k) index lookup, not a scan — `quote_orders_for_gateway` filters `orders_for_gateway`, which reads this gateway's resting orders directly from `_orders_by_gateway`. An earlier version of this function scanned `book.resting_orders()` (every resting order in the whole book) and filtered in Python for `gateway_id` and `origin == QUOTE`; that scan was replaced by this index once it became clear that a partial fill under `INACTIVATE_ON_ANY_FILL` — leaving a resting remainder with no `QuoteIndex` entry — is the ordinary case for an active market maker, not an edge case worth paying an O(n) scan for on every reissue.

3.13.7 Walkthrough: A Fill, a Stray Leg, and a Reissue



The stale 400-remainder is never orphaned indefinitely: it is genuinely tradeable until the moment the reissue arrives, at which point `quote_orders_for_gateway` finds it in O(1) and `_cancel_order_by_id` retires it — the same guarantee the design intended, at the complexity the engine's other indexes hold themselves to.

3.13.8 Restart: Both Indexes Rebuild From the Same Source

Neither index has its own on-disk format. `QuoteIndex` and `_orders_by_gateway` are both pure derivations from the persisted `Order` records (Section 13 covers GTC persistence in full). On restore, `Engine._restore_gtc` calls `book.process(order, match=False)` for every persisted order; whichever funnel that order's type reaches — `_rest()` for a resting `LIMIT`, `_add_stop()` for a `STOP`/`STOP_LIMIT`, `_add_trailing_stop()` for a `TRAILING_STOP` — populates `_orders_by_gateway` exactly as it would for a brand-new order, no separate rebuild code needed. `QuoteIndex` does need an explicit rebuild, because it groups legs by `quote_id` rather than being a pure by-product of resting one order: `_restore_gtc` collects every restored quote-origin order, groups them by `(gateway_id, quote_id)`, and re-inserts a `QuoteEntry` only for groups where *both* legs survived. A group with only one surviving leg is left as an ordinary resting order — logged as a "single-leg quote remnant" — rather than synthesizing a `QuoteEntry` for a quote that is missing half of itself.

3.14 11. The Message Bus: Why ZeroMQ?

3.14.1 The Problem with Direct Function Calls

Imagine implementing the exchange as a single monolithic program. The matching engine calls clearing functions directly; the viewer calls engine functions to get book state; the stats recorder calls engine functions to get trades. This works but has severe problems:

1. **Tight coupling**: the engine must know about clearing, the viewer, the stats recorder. Adding a new consumer requires modifying the engine.
2. **Single process**: a crash in the clearing code crashes the matching engine — catastrophic.
3. **Scalability**: you cannot run clearing on a different machine.
4. **Testing**: you cannot test the engine without starting all other components.

3.14.2 The Publish-Subscribe Pattern

EduMatcher uses a **message bus** — a communication channel where:

- The **engine publishes** events (trades, fills, book updates)
- Any number of **subscribers** receive the events they care about
- The engine has **no knowledge of its subscribers**

This is the ZeroMQ PUB/SUB pattern:

```
# messaging/bus.py

def make_publisher(addr: str) -> CausalPublisher:
    """PUB socket — engine binds here, broadcasts everything."""
    sock = get_context().socket(zmq.PUB)
    sock.bind(addr)
    # "A publisher" means one thing everywhere: sequenced, and causally
    # stamped. Callers that only publish need to know none of it.
    return CausalPublisher(SequencedPublisher(sock))

def make_subscriber(addr: str, *topics: str) -> zmq.Socket[bytes]:
    """SUB socket — subscriber connects and filters by topic prefix.
    No topics (or an empty string) subscribes to everything."""
    sock = get_context().socket(zmq.SUB)
    sock.connect(addr)
    if not topics:
        sock.setsockopt(zmq.SUBSCRIBE, b"")  # no filter — receive every topic
    else:
        for t in topics:
            sock.setsockopt(zmq.SUBSCRIBE, t.encode())
    return sock
```

The engine publishes on port 5556. Every other process connects to 5556 and subscribes to the topics it cares about. The engine does not know or care who is listening.

3.14.3 Topic-Based Filtering

ZeroMQ PUB/SUB filters by message prefix at the network level — subscribers only receive messages whose topic matches their subscription prefix. This means:

- The clearing process subscribes to `"trade.executed"` — receives only trades
- The stats process subscribes to `"trade.executed"` and `"book."` — trades and book snapshots
- The audit process subscribes to `""` (empty prefix) — receives everything
- A specific gateway subscribes to `"order.fill.GW01"` — only its own fills

```
# clearing/main.py
self.sub = make_subscriber(ENGINE_PUB_ADDR, "trade.executed")

# stats/main.py
self.sub = make_subscriber(ENGINE_PUB_ADDR, "trade.executed", "book.", "system.eod")

# audit/main.py
self.sub = make_subscriber(ENGINE_PUB_ADDR)  # no topics = subscribe to all
```

ZeroMQ performs topic matching in the PUB socket's send path before the bytes even hit the network. This means processes that subscribe to only one topic type see zero overhead from the thousands of other messages being published per second.

3.14.4 PUSH/PULL for Order Submission

Order submission uses a different ZeroMQ pattern — PUSH/PULL:

```
# engine binds a PULL socket
ENGINE_PULL_ADDR = "tcp://127.0.0.1:5555"
self.pull_sock = make_puller(ENGINE_PULL_ADDR)

# each gateway connects a PUSH socket
push_sock = make_pusher(ENGINE_PULL_ADDR)
push_sock.send_multipart(make_order_new_msg(order.to_dict()))
```

PUSH/PULL is a pipeline pattern: multiple pushers (gateways) deliver work to one puller (engine). ZeroMQ handles buffering and flow control. The engine processes one message at a time from its single PULL socket — this single-reader property is what makes the engine naturally thread-safe without any locks.

3.14.5 Message Format

`encode()`/`decode()` only ever look at the first two frames of a ZMQ message, and that is deliberate — it is what let two later features (a per-topic sequence number, then a causal-traceability envelope) ride along on every message without touching this function or any of its ~100 call sites:

```
Frame 0: topic bytes (e.g. b"order.fill.GW01")
Frame 1: JSON payload bytes (e.g. b'{"order_id": "...", "fill_qty": 100, ...}')
```

```
# models/message.py
def encode(topic: str, payload: dict[str, Any]) -> list[bytes]:
    """Return a two-frame ZMQ multipart message."""
    return [topic.encode(), _dumps(payload)]

def decode(frames: list[bytes]) -> tuple[str, dict[str, Any]]:
    """Parse a ZMQ multipart message.

    Reads only the topic and payload frames. Publishers append further
    frames (see below); they are ignored here so every subscriber keeps
    working unchanged.
    """
    topic   = frames[0].decode()
    payload = _loads(frames[1])
    return topic, payload
```

The frames behind the message: sequence and causal envelope

What actually goes out on the wire today has up to two more frames appended after the topic and payload — metadata *about* the message, not part of it, which is exactly why it lives in its own frames instead of extra JSON keys:

Frame	PUB (engine → subscribers)	PUSH (client → engine)
0	topic bytes	topic bytes
1	JSON payload	JSON payload
2	per-topic **sequence number** (`SequencedPublisher`)	**causal envelope** (`CausalPusher`)
3	**causal envelope** (`CausalPublisher`)	—

Two producers append these, composed in `make_publisher`/`make_pusher` (`messaging/bus.py`), and neither one is visible to `decode()`:

- **`SequencedPublisher`** inserts a per-topic counter at index 2 on every PUB message. ZeroMQ drops messages silently once a slow subscriber falls behind its high-water mark, and nothing in the payload reveals that it happened — the counter is what lets a subscriber notice a hole. It counts *per topic*, not per socket, so a subscriber that only takes `trade.executed` does not see a gap every time a `depth.*` message it never subscribed to goes by.

- **`CausalPublisher`** (PUB) / **`CausalPusher`** (PUSH) append a **causal envelope** — one ASCII frame, three `|`-separated fields: `msg_id|causation_id|correlation_id` (`models/envelope.py`). `msg_id` is a ULID (sortable by generation time) identifying this message; `causation_id` is the `msg_id` of the inbound request that caused it, or empty if nothing on the bus caused it; `correlation_id` names the whole causal chain, propagated unchanged from cause to effect. This is the traceability extension: it is what lets a tool answer *"which submission produced this fill, and everything else that followed from it?"* by looking the ids up, rather than inferring an answer from timing or from shared order ids. How a request's envelope propagates through the engine to everything it publishes while handling that request is covered later in this section (under Why ZeroMQ Specifically?) and in the Causal envelope reference in the architecture overview; how `pm-audit` records it in each log line is in the Audit Trail chapter.

Reading these frames back is symmetric with writing them, and neither reader raises on a message that does not have them — an older recording, or a test double standing in for a publisher, still parses:

```
# models/message.py
def decode_sequence(frames: list[bytes]) -> int | None:
    """The publisher's per-topic sequence number, or None if absent."""
    if len(frames) < 3:
        return None
    try:
        return int(frames[2])
    except (ValueError, TypeError):
        return None

def decode_envelope(frames: list[bytes]) -> Envelope | None:
    """The causal envelope, found by shape rather than by a fixed index.

    The envelope sits behind the sequence frame on PUB (index 3) but
    directly behind the payload on PUSH, which has no sequence (index 2).
    Scanning the trailing frames for whatever parses as an envelope removes
    the index entirely, rather than have two call sites disagree about it.
    """
    for frame in frames[2:]:
        env = Envelope.from_frame(frame)
        if env is not None:
            return env
    return None
```

Both frames are additive and append-only: `SequencedPublisher` inserts its counter *at* index 2 rather than appending, so that a wrapped `CausalPublisher` still lands its envelope frame last regardless of composition order, and every consumer that predates either feature — anything calling only `decode()` — is completely unaffected. That is also why the cost of the causal envelope is confined to the publish side: minting a ULID and formatting one short ASCII frame, never a JSON re-encode.

For JSON serialization, EduMatcher uses `orjson` when available — a C-extension library that is ~9-10x faster than the Python standard library `json`:

```
try:
    import orjson as _json_mod
    def _dumps(obj): return _json_mod.dumps(obj)  # returns bytes directly
    def _loads(data): return _json_mod.loads(data)
except ImportError:
    import json as _json_fallback
    def _dumps(obj): return _json_fallback.dumps(obj).encode()
    def _loads(data): return _json_fallback.loads(data)
```

At 2-4 encode/decode calls per order on the hot path, the difference between `json` (~2.1µs/call) and `orjson` (~0.22µs/call) saves 4-7µs per order — a significant fraction of the total latency budget.

3.14.6 Why ZeroMQ Specifically?

Several message bus technologies exist. ZeroMQ was chosen for EduMatcher because:

Brokerless — no central message broker process to manage. Each socket connects directly. This simplifies deployment and eliminates a single point of failure.

In-process or cross-network — switching from `tcp://127.0.0.1:5555` to `tcp://10.0.1.5:5555` requires changing a single string. The same code works for local processes and distributed systems.

Extremely low latency — ZeroMQ message passing overhead is in the microseconds. For a trading system where each additional millisecond matters, this is essential.

Simple API — a few lines to create a publisher, three to create a subscriber.

The module is small, but two of its wrappers are worth knowing about, because they are how the exchange answers "why did this happen?" without anyone having to remember to record it.

`SequencedPublisher` appends a per-topic counter to every published message. PUB/SUB drops silently when a subscriber falls behind; the counter is what lets a subscriber notice. Counting per topic rather than per socket means a subscriber filtering on `trade.` does not see phantom gaps from every `depth.` message it filtered out.

`CausalPublisher` / **`CausalPusher`** stamp the causal envelope — `msg_id`, `causation_id`, `correlation_id` (see `models/envelope.py`). A gateway's request starts a chain; everything the engine publishes while handling that request cites it and inherits the chain:

```
# engine receive loop — the whole of the causality plumbing
frames = self.pull_sock.recv_multipart()
topic, payload = decode(frames)
cause = decode_push_envelope(frames)

self.pub_sock.set_cause(cause)
try:
    self._dispatch_pull_message(topic, payload)
finally:
    self.pub_sock.clear_cause()   # never leak attribution to the next message
```

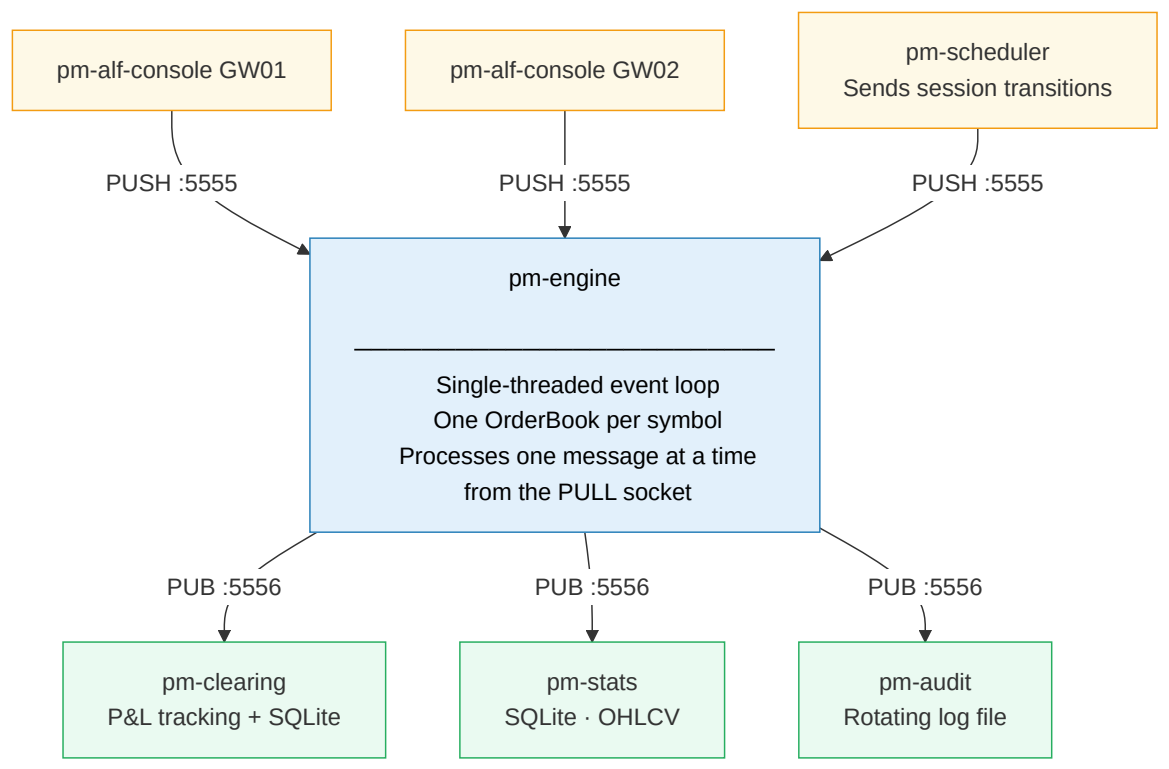
Wrapping the socket rather than editing each publish site is the point: the engine publishes from roughly a hundred places, and an envelope attached only where someone remembered would be worse than none — a missing `causation_id` is supposed to mean "nothing caused this", so a forgotten one would be a lie rather than a gap.

The `finally` matters more than it looks. Without it, a handler that raises would leave the cause set, and the next messages — including maintenance work with no external cause at all — would be attributed to a request that had already failed.

Zero configuration — no broker process, no configuration files, no schema registry. Just connect and start sending.

3.15 12. The Full Process Architecture

EduMatcher is a **multi-process system**. Each process is a separate Python program with its own memory space, started independently. The diagram below shows the core message-flow processes. Display processes (`pm-viewer`, `pm-board`, `pm-ticker`) also connect to the PUB socket on port 5556 but are omitted for clarity — they are pure subscribers that display data and never send orders to the engine. The AI traders (`pm-ai-trader`, `pm-ai-swarm`) read the same PUB socket but send their orders as ALF clients through `pm-alf-gwy`, like a student's console; the market model (`pm-market-sim`) has PUB/PULL sockets of its own on ports 5553/5554.



3.15.1 The Engine: Single-Threaded by Design

The matching engine uses a single-threaded **event loop** — a programming pattern where a program continuously waits for the next event (a message, a timeout, a signal), processes it fully, then returns to waiting. No work happens in parallel; everything is sequential. This is not a limitation — it is a deliberate design choice:

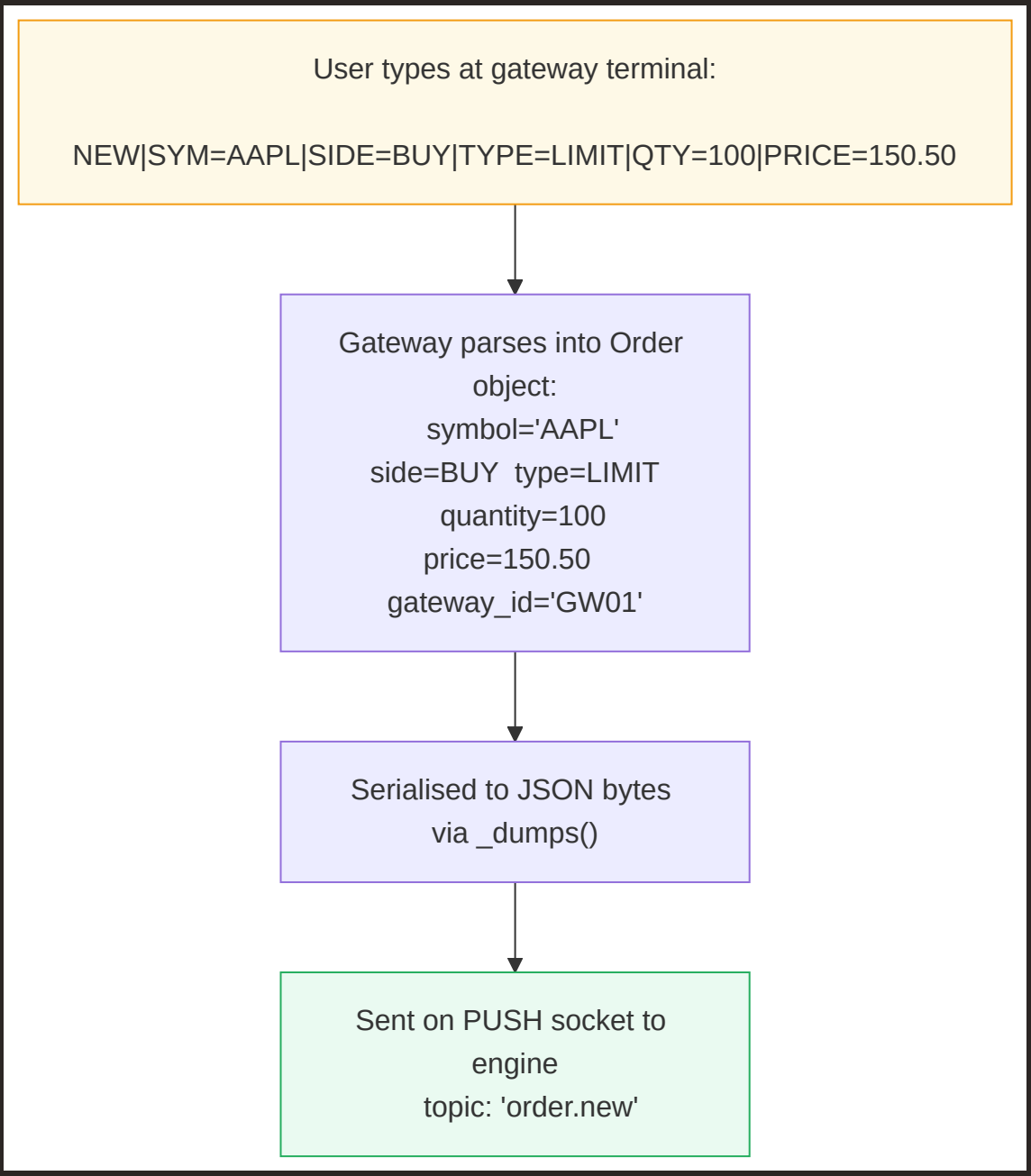
```
# engine/main.py
while self._running:
    socks = dict(poller.poll(timeout=200))  # wait up to 200ms for a message
    if self.pull_sock in socks:
        frames = self.pull_sock.recv_multipart()
        topic, payload = decode(frames)
        if topic == "order.new":
            self._handle_new_order(payload)
        elif topic == "order.cancel":
            self._handle_cancel(payload)
        # ... etc
    self._flush_snapshots()  # throttled book snapshot publishing
```

Because only one message is processed at a time, there are **no race conditions**. Every data structure in the engine — `self.books`, `self._order_symbol`, `self._combos`, `self._oco_groups` — is modified by a single thread. No locks are needed. No atomic operations. No memory barriers.

This is a well-known pattern in financial systems: a **single-writer** architecture where correctness is ensured by sequential processing rather than synchronisation. The cost is that the engine cannot use multiple CPU cores for matching — but in practice, a single modern CPU core can process tens of thousands of orders per second, more than sufficient for most exchange volumes.

3.15.2 The Gateway: User Interface

The gateway translates human-readable ALF commands into ZMQ messages (see ALF Protocol Reference):



The gateway has two threads:

- **Main thread**: reads commands from stdin (with readline/prompt_toolkit), sends to engine
- **Subscriber thread**: listens for responses from the engine (acks, fills, cancellations), displays them to the user

These threads communicate via ZMQ rather than shared memory, keeping them decoupled.

3.15.3 Clearing: Tracking Positions and P&L

The real position state lives in a private `_Position` dataclass in `clearing/ledger.py` (field names differ — `net_qty`, `mark_price`, etc.); the conceptually-equivalent, simplified version below is easier to read fill logic against (Section 15 covers the full real VWAP walkthrough):

```
# conceptually clearing/ledger.py — simplified field names
@dataclass
class PositionRecord:
    symbol:       str
    gateway_id:   str
    position:     float = 0.0   # net qty (+ = long, - = short)
    avg_cost:     float = 0.0   # VWAP cost basis
    realized_pnl: float = 0.0   # from closed legs
    last_price:   float = 0.0

    @property
    def unrealized_pnl(self) -> float:
        return self.position * (self.last_price - self.avg_cost)
```

The clearing process subscribes to `trade.executed`, updates a P&L ledger for each participant on each side of every trade, and batches the updated rows into `clearing.db` (SQLite) — never a CSV file; see Section 15 for the history of the `clearing_report.csv` path this replaced. The VWAP cost basis update:

```
def apply_fill(self, qty: int, price: float, is_buy: bool) -> None:
    signed_qty = qty if is_buy else -qty
    if (self.position > 0 and is_buy) or (self.position < 0 and not is_buy):
        # Adding to position — volume-weighted average price update
        total_cost = self.avg_cost * abs(self.position) + price * qty
        self.position += signed_qty
        self.avg_cost = total_cost / abs(self.position)
    else:
        # Reducing position — realise P&L at the difference from avg cost
        reduce_qty = min(qty, abs(self.position))
        self.realized_pnl += (price - self.avg_cost) * reduce_qty * (1 if self.position > 0 else -1)
        self.position += signed_qty
```

3.15.4 Stats: OHLCV and Market Data

The stats process subscribes to trades and book snapshots, persisting them to SQLite:

- `trade_log` — every individual trade (price, quantity, buyer, seller)
- `daily_stats` — one row per (date, symbol): open, high, low, close, volume, VWAP, trade count
- `price_snapshots` — mid-price snapshot every 15 minutes per symbol

3.15.5 Audit: The Complete Event Log

The audit process subscribes to **all topics** (empty filter):

```
self.sub = make_subscriber(ENGINE_PUB_ADDR)  # subscribes to everything
```

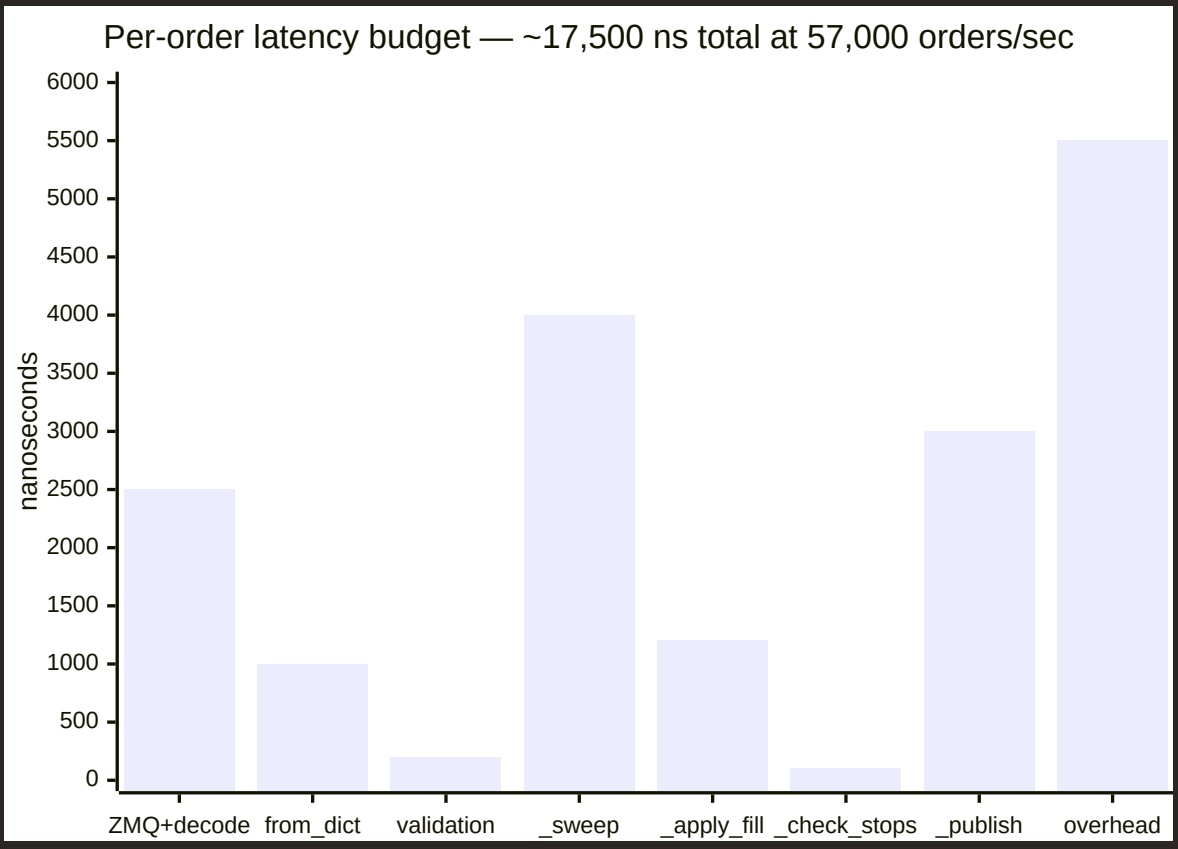
Every event — order acks, fills, cancellations, trades, book snapshots, session transitions — is written as a timestamped JSON line to a rotating log file. This is the closest thing to a full audit trail in the system.

3.16 13. Optimisations: Speed, Memory, and Latency

This section collects all performance-oriented decisions in one place, explains the reasoning, and quantifies the expected impact where possible.

3.16.1 The Latency Budget

Before reading individual optimisations, it helps to understand what we are optimising for. At the tested throughput of 57,000 orders/second, each order has ~**17.5 microseconds** of budget (1 second ÷ 57,000):



Step	Component	Cost	Notes
1	ZMQ receive + decode JSON	~2,500 ns	
2	`Order.from_dict()`	~1,000 ns	With `__new__` bypass + enum maps
3	Tick validation + price conversion	~200 ns	
4	`_sweep()` matching loop	~4,000 ns	Varies by fill count
5	`Trade.create()` + `_apply_fill()`	~1,200 ns	
6	`_check_stops()`	~100 ns	O(1) when no stops trigger
7	`_publish_events()` + `_publish_trades()`	~3,000 ns	JSON encode + ZMQ send
8	Overhead (function calls, misc)	~5,500 ns	
	Total	**~17,500 ns**	

Each optimisation in this section attacks one of these buckets. The numbers are approximate and vary by hardware, OS scheduler, and fill rate — but the relative proportions are stable. JSON serialization and the sweep loop together consume the most time; both receive dedicated optimisations below.

3.16.2 Architecture: Single Timestamp Per Order

```
# engine/main.py — _handle_new_order()
now = now_ns()  # ONE call per incoming order — int nanoseconds
                    # from models/clock.py; guaranteed strictly increasing

trades, events = book.process(order, match=do_match, now=now)
# now is threaded through:
#   _match_limit → _sweep → _apply_fill → Trade.create(now=now)
#   _check_stops → triggered.timestamp = now
#   _reinsert_iceberg → iceberg.timestamp = now
```

`now_ns()` is a wrapper around `time.time_ns()` that guarantees strictly increasing values even when the system clock steps

backward (e.g. due to NTP adjustments on a virtual machine). Each call costs ~300-500ns on macOS/Linux. An aggressive order that triggers stop orders calls `process()` again for each released stop (an iterative work queue, not recursion). Without this optimisation, each of those calls would call `now_ns()` independently. With it: one call, threaded through as a parameter.

Why do triggered stops need the same `now` as the original order?

When an aggressive order fills and triggers a stop, the stop order enters the book with `timestamp = now`. If the triggered order were assigned a fresh `now_ns()` timestamp, it would appear to have arrived *later* than the fill that triggered it — which is technically accurate but creates a subtle inconsistency: all the events produced by one incoming message would have slightly different timestamps, making the audit log harder to reason about. Using the same `now` for the entire cascade means "all events caused by message X happened at time T."

Savings: 2-4 `now_ns()` calls eliminated per aggressive order with stops 0.6-2µs per order.

3.16.3 `__slots__` on All Hot-Path Objects

Applied to: `Order`, `Trade`, `_HeapEntry`, `OrderBook`.

The reasoning is the same for all: these are the objects accessed most frequently in the inner loop. `__slots__` eliminates the per-object `__dict__`, making attribute access ~30% faster and reducing memory by ~40% per instance. This also reduces GC (Garbage Collection) pressure — with fewer and smaller objects allocated per order, Python's garbage collector runs less frequently.

At 57,000 orders/second (the tested throughput), `__slots__` on `Order` alone saves roughly 2µs per order from reduced attribute lookup in the sweep loop.

3.16.4 Pre-Built Enum Lookup Dicts

```
_SIDE_MAP:   dict[str, Side]   = {v.value: v for v in Side}
_TYPE_MAP:   dict[str, OrderType] = {v.value: v for v in OrderType}
# ... etc
```

Python's `Enum("BUY")` constructor iterates all members doing string comparisons. For a 5-member enum this is ~600-800ns. A dict lookup is ~50ns. With 5 enum constructions per `Order.from_dict()`, this saves ~3-4µs per inbound order.

3.16.5 `from_dict` via `__new__` Bypass

```
o = object.__new__(cls)     # skip __init__ entirely
o.id = d["id"]              # write slots directly
o.side = _SIDE_MAP[d["side"]]
# ...
```

The dataclass-generated `__init__` with 26 keyword arguments spends ~400ns on Python's argument parsing and dispatch machinery. Bypassing it via `__new__` and direct slot writes reduces `from_dict` from ~1400ns to ~1000ns — a 29% improvement on the deserialization hot path.

3.16.6 Pre-Cached ZMQ Topic Bytes

```
# engine/main.py
self._topic_cache: dict[str, bytes] = {}

# On first order from a gateway, populate all hot topics:
_tc[_gw]               = f"order.ack.{_gw}".encode()
_tc[f"fill.{_gw}"]     = f"order.fill.{_gw}".encode()
_tc[f"cancel.{_gw}"]   = f"order.cancelled.{_gw}".encode()

# On every subsequent order, use the cached bytes:
self.pub_sock.send_multipart([
    _tc.get(f"fill.{evt.gateway_id}") or f"order.fill.{evt.gateway_id}".encode(),
    _dumps({...}),
])
```

Building a ZMQ topic bytes object requires an f-string format operation plus `.encode()`. Cost: ~100ns per message. With 3-4 messages per order and thousands of orders per second, pre-caching saves ~300-400ns per order.

3.16.7 Inlined Message Dicts on Hot Paths

The `make_fill_msg()` helper function builds a base dict then calls `.update()` to merge order fields. Two allocations plus a hash merge. For high-frequency paths, EduMatcher inlines the entire dict literal:

```
# Instead of:
self.pub_sock.send_multipart(make_fill_msg(gw, order_id, fill_qty, ...))

# The engine uses (prices converted from int ticks to float at this output boundary):
from edumatcher.models.price import from_ticks

self.pub_sock.send_multipart([
    fill_topic_bytes,           # pre-cached
    _dumps({                    # single allocation, single orjson call
        "order_id":      evt.id,
        "fill_qty":      evt.quantity - evt.remaining_qty,
        "fill_price":    from_ticks(book.last_trade_price, evt.symbol),
        "remaining_qty": evt.remaining_qty,
        "status":        evt.status.value,
        "symbol":        evt.symbol,
        "side":          evt.side.value,
        "order_type":    evt.order_type.value,
        "qty":           evt.quantity,
        "price":         from_ticks(evt.price, evt.symbol) if evt.price else None,
    }),
])
```

The helper function is preserved in `message.py` for use in non-hot paths (testing, less-frequent operations) where readability matters more than speed.

3.16.8 Throttled Book Snapshot Publishing

Publishing a full order book snapshot requires iterating all heap entries — O(n). Doing this on every fill at high throughput would consume significant CPU on serialization alone.

```
SNAPSHOT_INTERVAL = 0.5  # seconds

def _flush_snapshots(self) -> None:
    # time.monotonic() here is correct and intentional. We are measuring
    # elapsed wall-clock duration (how many seconds since the last snapshot),
    # NOT generating an event timestamp for price-time priority. For duration
    # measurements, time.monotonic() returning a float of seconds is exactly
    # right. now_ns() from models/clock.py is only for event timestamps.
    now = time.monotonic()
    sent: set[str] = set()
    for symbol in self._dirty_symbols:
        last = self._last_snapshot.get(symbol, 0.0)
        if now - last >= self.SNAPSHOT_INTERVAL:
            book = self.books.get(symbol)
            if book:
                self.pub_sock.send_multipart(make_book_msg(symbol, book.snapshot()))
            self._last_snapshot[symbol] = now
            sent.add(symbol)
    self._dirty_symbols -= sent
```

Books are marked “dirty” when any order changes their state. Snapshots are published at most every 500ms per symbol, regardless of fill rate. This decouples snapshot publishing cost from order throughput.

3.16.9 Trade ID: Monotonic Counter Instead of UUID

```
# models/trade.py
_trade_counter = itertools.count(1)

@classmethod
def create(cls, ...):
    trade_seq = next(_trade_counter)
    return cls(
        id=f"{_run_seq:06d}-{trade_seq:09d}",  # ≈50ns plus formatting
        ...
    )
```

`uuid.uuid4()` calls `/dev/urandom` — a system call costing ~1.5-2µs. Trade IDs combine a persisted engine-run sequence with an in-process monotonic counter, so they stay durable and sortable without a syscall in the inner loop. At 57,000 trades per second, avoiding UUID generation saves roughly 86ms per second of CPU time on ID generation alone.

3.16.10 Local Variable Caching in the Inner Loop

```
def _sweep(self, aggressor, opposite_heap, price_limit, trades, events, now):
    # Cache before the loop — not inside it
    _side       = aggressor.side        # avoid re-reading slot on each iteration
    _smp_action = aggressor.smp_action
    _gw_id      = aggressor.gateway_id
    _peek       = self._peek            # avoid method lookup on each call
    _apply_fill = self._apply_fill

    while aggressor.remaining_qty > 0 and opposite_heap:
        best = _peek(opposite_heap)  # local call — no self. lookup
        if _side == Side.BUY and best.price > price_limit:
            break
        ...
```

In CPython, `self.attribute` is a `LOAD_ATTR` bytecode instruction (descriptor lookup, ~50-70ns). A local variable is `LOAD_FAST` (array index, ~30ns). In a loop that may iterate 10+ times for a single aggressive order sweeping multiple price levels, caching saves ~20-40ns per iteration × N iterations = 0.2-0.8µs per order.

This is standard practice in performance-sensitive Python and is documented in the Python documentation as a recommended optimisation for hot loops.

3.16.11 `orjson` for Serialization

```
try:
    import orjson as _json_mod
    def _dumps(obj): return _json_mod.dumps(obj)  # ~0.22µs
except ImportError:
    import json as _json_fallback
    def _dumps(obj): return _json_fallback.dumps(obj).encode()  # ~2.1µs
```

`orjson` is a C-extension library with 9-10x better throughput than `json`. With 2-4 encode calls per order on the hot path, this saves 4-7µs per order at typical trade rates.

3.16.12 FOK Pre-Check

For a FOK order that cannot be filled, a naive engine would start filling orders (generating events), realise partway through it cannot complete, and need to reverse those fills — extremely complex. Instead `_fok_available_qty` computes, before any fill, how much the order could really execute against: the live resting orders in price-time order up to the limit price, including iceberg reserve, minus what self-match prevention would skip. Rejection is therefore clean. (The `_bid_qty` / `_ask_qty` price-level index is cheaper, O(P), but holds only visible quantity, so it is used for depth and best bid/offer, not for this.)

3.17 14. Persistence: Surviving a Restart

When the engine restarts, two categories of state need to be restored:

GTC and DAY Orders — resting limit orders with `TIF.GTC` survive trading days, as expected. Less obvious: resting `TIF.DAY` orders are serialised too, because a process restart is **not itself a day boundary** — only a genuine calendar rollover ends a DAY order's validity. On shutdown both are written as a plain JSON array of `Order.to_dict()` entries, via a helper that writes to a temp file and renames it into place so a crash mid-write can never leave a truncated, unreadable file:

```
# engine/persistence.py
def save_gtc_orders(orders: list[Order], path: Path) -> None:
    gtc = [
        o.to_dict()
        for o in orders
        if o.tif in (TIF.GTC, TIF.DAY)
        and o.status in (OrderStatus.NEW, OrderStatus.PARTIAL)
    ]
    _atomic_write_text(path, json.dumps(gtc, indent=2))  # temp file + rename
    # → data/gtc_orders.json
```

Loading is deliberately fail-soft, not fail-hard: a missing or unparseable file yields an empty list (the engine just starts with no resting orders) rather than crashing, and an individual corrupt entry is logged and skipped so the rest of the file can still be restored:

```
def load_gtc_orders(path: Path) -> list[Order]:
    if not path.exists():
        return []
    try:
        data = json.loads(path.read_text())
    except Exception:
        return []  # truncated / corrupt — start with no resting orders
    orders = []
    for d in data:
        try:
            orders.append(Order.from_dict(d))
        except Exception:
            pass  # logged at CRITICAL and skipped; other entries still load
    return orders
```

On the next startup, they are restored before any new orders are accepted. A restored `TIF.DAY` order is discarded instead if its original business day (derived from its `ts_ns`) is before today's date — the calendar has rolled over while it was resting, so it is no longer valid even though the process restart itself didn't end it:

```
def _restore_gtc(self) -> None:
    orders = load_gtc_orders(GTC_ORDERS_FILE)
    for order in orders:
        if order.tif == TIF.DAY:
            order_day = datetime.fromtimestamp(order.ts_ns / 1e9).date()
            if order_day < today:
                continue  # stale — business day has rolled over, discard
        order.status = OrderStatus.NEW  # reset from PARTIAL if needed
        book = self._book(order.symbol)
        book.process(order)  # re-insert into the heap
        self._order_symbol[order.id] = order.symbol
```

The original `ts_ns` is preserved in the serialisation (`Order.to_dict()` includes it). This means restored orders maintain their original price-time priority — a participant who submitted an order before the restart still gets their earlier position in the queue.

Book Statistics — the last traded prices per symbol, plus the previous close, are persisted so that stop order logic and display have a reference price on restart:

```
def save_book_stats(books: dict[str, OrderBook], path: Path) -> None:
    stats = {
        symbol: {
            "last_buy_price":  book.last_buy_price,
            "last_sell_price": book.last_sell_price,
            "prev_close":      book.last_trade_price,  # from_ticks()'d for display
        }
        for symbol, book in books.items()
    }
    _atomic_write_text(path, json.dumps(stats, indent=2))
    # → data/book_stats.json
```

GTC Combos — multi-leg combo orders whose legs are resting GTC are also persisted, along with the parent-child linkage needed to continue tracking them after restart.

3.17.1 What Persistence Does NOT Cover

- **Trade history** — the clearing CSV is the trade record, not the engine. The engine has no trade log beyond the in-memory `recent_trades` deque (maxlen=20) per book.
- **Positions** — clearing maintains positions in-memory. A clearing process restart loses all position state (cleared from the ledger in memory).

3.18 15. Clearing and Settlement: Following the Money

Clearing is the process of ensuring both sides of a trade actually deliver what they promised: the buyer delivers money, the seller delivers shares. EduMatcher implements a simplified clearing process that tracks positions and P&L.

```
# clearing/main.py
class ClearingProcess:
    def __init__(self):
        # ledger[gateway_id][symbol] → PositionRecord
        self._ledger: dict[str, dict[str, PositionRecord]] = defaultdict(dict)
```

When a trade arrives:

1. The buyer's position increases by `trade.quantity`
2. The seller's position decreases by `trade.quantity`
3. If this closes or reduces a position, realised P&L is computed
4. The trade and the updated position/daily rows are batched and flushed to `clearing.db` (SQLite) — not to a CSV file. `clearing_report.csv` was v1's mechanism before the SQLite-first redesign; the path, and its reference in `pm-opctl-cli clear`'s deletion list, have both been removed. For an ad-hoc CSV export, use `pm-clearing-cli ... --format csv`.

The P&L calculation uses **VWAP (Volume-Weighted Average Price)** as the cost basis. If you buy 100 shares at $150 and then 50 more at $160, your average cost is (100×150 + 50×160) / 150 = $153.33. When you later sell, your profit is computed against this average, not against any specific purchase.

Concrete P&L walkthrough:

```
Trade 1: BUY  100 AAPL at $150.00
  position = +100,  avg_cost = $150.00,  realized_pnl = $0

Trade 2: BUY   50 AAPL at $160.00  (adding to the long position)
  total_cost = 150.00 × 100 + 160.00 × 50 = $23,000
  position   = 150
  avg_cost   = 23,000 / 150 = $153.33
  realized_pnl = $0   (no reduction in position yet)

Trade 3: SELL  80 AAPL at $165.00  (partially closing)
  reduce_qty   = min(80, 150) = 80
  realized_pnl += (165.00 - 153.333333) × 80 = $933.33   (using the unrounded
                                    avg_cost; 153.33 above is display-rounded)
  position     = 150 - 80 = 70     (still long 70 shares)
  avg_cost     = $153.33            (unchanged — we closed at market, avg cost of
                                     remaining position is still the VWAP)

  unrealized_pnl = 70 × (165.00 - 153.33) = $816.67 (if last_price = $165.00)
```

```
def apply_fill(self, qty, price, is_buy):
    signed_qty = qty if is_buy else -qty
    if (self.position > 0 and is_buy) or (self.position < 0 and not is_buy):
        # Adding to an existing position (long getting longer, or short getting shorter).
        # Update the VWAP average cost to include the new fill.
        total_cost = self.avg_cost * abs(self.position) + price * qty
        self.position += signed_qty
        self.avg_cost = total_cost / abs(self.position)
    else:
        # Reducing or reversing a position — realise P&L on the closed portion.
        reduce_qty = min(qty, abs(self.position))
        self.realized_pnl += (price - self.avg_cost) * reduce_qty * (1 if self.position > 0 else -1)
        self.position += signed_qty
        # Note: if the fill crosses zero (e.g. long 50, sell 100 → short 50),
        # avg_cost should be reset to the fill price for the new short portion.
        # This simplified version does not handle that case.
```

In a real exchange, clearing is far more complex:

- **CCP (Central Counterparty) novation** — when a trade is novated, the CCP legally steps between the buyer and seller, becoming the seller to every buyer and the buyer to every seller. This means if one side defaults, the CCP absorbs the loss rather than the counterparty. Novation requires the CCP to have enough capital to cover potential defaults.
- **Margin requirements** — participants must deposit collateral (margin) to cover their potential losses. The CCP calculates and collects this daily.
- **Settlement (T+2 delivery)** — "T+2" means the actual transfer of shares and money happens two business days after the trade date (Trade date + 2). This delay exists because settlement systems need time to confirm and coordinate the transfers.
- **Netting** — rather than settling each trade individually, a participant's many buys and sells in the same stock across the day are aggregated. Only the net position change is settled, dramatically reducing the number of actual transfers.

EduMatcher's clearing is educational — it shows the concept without the regulatory and legal infrastructure.

3.19 16. What Is Missing: The Gap to Production

EduMatcher is explicitly educational. Here is a complete and honest accounting of what would need to be added to make it production-grade. Each item is a non-trivial engineering project.

3.19.1 Guaranteed Delivery and Replay on the Core Message Bus

What already exists: every message the engine publishes on its internal ZMQ PUB sockets *does* carry a monotonically increasing, per-topic sequence number — `SequencedPublisher` (`messaging/bus.py`) stamps it as a third wire frame, and `decode_sequence()` reads it back. Sequences start at 1 per topic and reset on process restart (a decrease signals a restart, not

a gap), so a subscriber that tracks the last sequence it saw per topic *can* detect a hole in what it received — the core "is something missing?" problem is already solved. A narrower, bounded form of replay also already exists one layer up: the WebSocket API gateway's market-data stream supports a client-sent `resume/from_seq` control that re-sends buffered `trades` from its cache (see REST and WebSocket) — but that cache is bounded and in-memory, and covers only the trade tail of one WebSocket stream, not the engine's internal bus.

What is still missing: on the core ZMQ bus (the one clearing, stats, and audit consume), there is no `replay_request`/response protocol and no persistent message log. If one of those processes disconnects and reconnects, it can now *notice* a gap via the sequence numbers, but there is nothing to ask for the missing messages back — PUB/SUB has already dropped them, and nothing durable stored them for re-sending.

What production requires: a persistent, bounded replay buffer (or a full commit log, as in Apache Kafka) behind the core bus, and a request/response protocol so clearing/stats/audit can ask for messages from a given sequence forward after a reconnect — generalising the pattern the WS market-data stream already uses for trades to the whole bus, and backing it with durable storage rather than an in-memory cache.

```
MISSING (core ZMQ bus only — sequence numbers already exist):
  - Replay request/response protocol
  - Persistent message log with configurable retention
  - Subscriber reconnect logic with automatic replay
```

3.19.2 Authentication and Authorisation

What is missing: Any process can connect to port 5555 and submit orders as any `gateway_id`. There is no proof that `GW01` is actually GW01.

The existing `system.gateway_connect` handshake:

```
# engine/main.py — after Phase 2 of the implementation plan
def _handle_gateway_connect(self, payload):
    gateway_id = payload.get("gateway_id")
    self._sessions[gateway_id] = ParticipantSession(
        gateway_id=gateway_id,
        role=ParticipantRole.TRADER,   # default unless configured otherwise
        connected=True,
    )
    self.pub_sock.send_multipart(make_gateway_auth_msg(gateway_id, accepted=True))
```

This accepts any connection that knows the port number. `ParticipantSession` is a dataclass that records each connected gateway's role (`ParticipantRole.TRADER`, `ParticipantRole.MARKET_MAKER`, or `ParticipantRole.ADMIN`) and its disconnect behaviour (whether to cancel its orders or leave them resting when the gateway disconnects). These roles control what message types the gateway may submit (e.g. only `MARKET_MAKER` sessions may submit two-sided quotes). However, there is no cryptographic proof that the connecting process is who it claims to be.

What production requires:

- **Authentication**: cryptographic proof of identity. In practice: TLS client certificates, or a challenge-response protocol where the client proves knowledge of a private key.
- **Authorisation**: the authenticated identity is mapped to permissions. GW01 may trade AAPL and MSFT but not TSLA. MM01 may submit quotes. ADMIN01 may cancel any order. These permissions must be checked on every operation.
- **Session tokens**: short-lived tokens issued after authentication, rotated periodically.
- **Audit trail**: every authentication event and authorisation decision logged.

3.19.3 TLS Encryption

What is missing: All ZMQ communication is plaintext TCP. Any process on the network can read order messages, trade data, and fill reports.

What production requires: TLS (Transport Layer Security — the same encryption that secures HTTPS web traffic) or ZMQ's built-in **CurveZMQ** (a lightweight elliptic-curve cryptography layer built directly into ZeroMQ, which provides both encryption and authentication without a full TLS certificate infrastructure) encryption on all sockets.

3.19.4 Primary and Secondary (Failover) Sites

What is missing: EduMatcher runs on one machine. If that machine crashes, the exchange stops. All in-flight orders and state are lost.

What production requires: A complete standby site running a **replica** of the primary engine. Replication strategies include:

Active-Passive: The primary engine processes all orders. Every message is also sent to the secondary, which replays them but does not publish results. If the primary fails, the secondary activates and takes over from its last replicated state. The challenge: at the moment of failover, messages in-flight (sent to primary but not yet acknowledged to gateways) may be processed twice on the secondary (duplicate orders) or not at all (lost fills). This requires careful sequencing.

Active-Active: Both engines process orders, and a consensus protocol (such as **Raft** or **Paxos** — distributed algorithms that allow a group of machines to agree on a sequence of decisions even if some machines fail) ensures they agree on every fill before publishing. Slower but provides no loss. Used by the most critical systems (e.g. central clearing houses).

Log-based replication: Every order message is written to a durable log (e.g. **Apache Kafka** — a distributed event streaming platform that stores messages in an ordered, persistent, replicated log) before being processed. The secondary replays from the log. Failover is a matter of the secondary catching up to the primary's position in the log and activating. Clean and auditable.

The specific challenge for an exchange: during failover, gateways must reconnect to the secondary site without losing their session state (open orders, pending fills). This requires the secondary to have a complete, up-to-date copy of all open order state.

```
MISSING:
  - Engine state replication
  - Failover detection (heartbeat monitoring)
  - Failover activation procedure
  - Gateway reconnect during failover
  - Duplicate message detection after failover
```

3.19.5 Order Replay and Recovery

What is missing: If the engine crashes mid-processing (after receiving an order but before publishing its ack), that order is silently lost. The gateway never receives an ack or rejection — it is left in an unknown state.

What production requires: A **write-ahead log (WAL)** — before processing any order, the engine writes it to a durable log. On restart, the engine replays the log to reconstruct the exact order book state up to the crash point. Gateways that reconnect receive replayed acks for any orders that were processed before the crash.

This is similar to how database transaction logs work. The core principle: if something is committed to the log, it happened; if it is not in the log, it did not.

3.19.6 Rate Limiting and Risk Checks

What is implemented:

- **Price collars** — static and dynamic band checks that reject any order whose price deviates by more than a configured percentage from the last trade price or a reference price. Configured per symbol in `engine_config.yaml`.
- **Circuit breakers** — multi-level halt mechanism. When a trade price moves beyond configured thresholds within a time window, the symbol is halted. Configurable resumption modes include `AUCTION` (runs a local uncross before resuming).
- **Kill switch** — immediate cancellation of all resting orders (or all orders for a specific symbol) for a gateway, invokable via the `risk.kill_switch` message.

What is still missing:

- **Rate limits**: any gateway can submit an unlimited number of orders per second. A buggy algorithm could overwhelm the engine. Production requires per-gateway message rate enforcement at the gateway level.
- **Fat finger protection**: "fat finger" is trading slang for accidentally typing the wrong number (e.g. selling 10,000 shares when you meant 100, or pricing at $15.00 when you meant $150.00). Fat finger filters reject orders whose price or quantity deviates wildly from expected ranges. The existing collars partially address this but do not cover maximum order size or notional value checks.

3.19.7 High-Precision Timestamps and Clock Synchronisation

Status in EduMatcher: Nanosecond integer timestamps via `now_ns()` (from `models/clock.py`) are now implemented — see `EduMatcher_Tick_Migration_Plan_v6.md`. `now_ns()` wraps `time.time_ns()` with a guarantee of strictly increasing values even when the system clock is adjusted backward. All `Order`, `Trade`, and `ComboOrder` timestamps are `int` nanoseconds. The price-time priority heap key is actually `(±price_ticks, arrival_seq)` (Section 6) — an engine-assigned monotonic sequence, not the timestamp itself, so a back-dated client timestamp cannot jump the queue; both components are integers either way, so comparison is exact.

What still needs production work:

- **PTP (IEEE 1588 Precision Time Protocol)** — a standard for synchronising clocks across a network to sub-microsecond accuracy. Without PTP, timestamps from different machines can differ by milliseconds or more, making cross-process timestamp comparisons unreliable. GW01's timestamp and the engine's received-at timestamp may diverge significantly if their system clocks are not synchronised.
- **Timestamping at the wire** — hardware timestamps captured in the network card at the moment a packet arrives, rather than when software processes it. Required for regulatory compliance in most jurisdictions, which mandate proof of when an order was received to the microsecond.

3.19.8 Price Precision

Status in EduMatcher: Integer tick-based prices are now implemented — see `EduMatcher_Tick_Migration_Plan_v6.md`. All prices inside the engine are stored as `int` tick counts. For a symbol with `tick_decimals=2` (the default), `$150.30` is stored as `15030`. Conversion to/from float happens only at I/O boundaries (gateway input and ZMQ output). This eliminates floating-point equality bugs in matching and dict key collisions in the price-level index.

The original problem and why it was worth solving:

```
>>> 100.1 + 0.2
100.30000000000001  # not 100.30
>>> 150.30 == 150.10 + 0.20
False               # should be True — would cause a missed match in old code
```

What still needs production work: The tick size per symbol is configured statically in `engine_config.yaml`. A production system needs dynamic tick size changes (e.g. when a stock splits), per-exchange tick tables, and validation that submitted prices are multiples of the tick size before they enter the engine.

3.19.9 Network Level: FIX Protocol

What is missing: EduMatcher's gateway uses a simplified text protocol (`NEW|SYM=AAPL|SIDE=BUY|...`). Real exchanges use the **FIX (Financial Information eXchange) protocol** — an industry standard for order submission that has been in use since 1992.

FIX is a tag-value format:

```
8=FIX.4.4|35=D|49=BROKER01|56=EXCHANGE|11=ORD001|55=AAPL|54=1|38=100|44=150.50|40=2|
```

Tag 8 = protocol version, 35 = message type (D = new order single), 55 = symbol, 54 = side (1 = buy), 38 = quantity, 44 = price, 40 = order type (2 = limit).

Supporting FIX requires: a FIX engine (parser, session management, sequence number tracking, resend request handling), support for multiple FIX versions (4.2, 4.4, 5.0), and extensions for specific asset classes and order types.

3.19.10 Regulatory Requirements

What is missing: Depending on jurisdiction, an exchange must:

- **Report all trades** to a trade repository within milliseconds. **MiFID II** (Markets in Financial Instruments Directive II — the EU's primary regulation governing trading venues, in force since 2018) and **Regulation NMS** (National Market System — the US regulation requiring best execution across all US equity exchanges) both mandate near-real-time trade reporting.
- **Maintain audit logs** for 5-7 years, accessible on regulatory demand
- **Implement best execution** policies — document that orders were filled at the best available price
- **Market surveillance** — detect and report potentially manipulative patterns:

  - **Spoofing**: placing large orders to create the appearance of demand or supply, then cancelling them before they execute — intended to mislead other participants about where prices are headed
  - **Layering**: a form of spoofing using multiple orders at several price levels to create a false order book impression
  - **Wash trading**: buying and selling the same instrument to yourself (or a related party) to generate artificial volume — detected by SMP but requiring broader surveillance to catch across different accounts
- **Circuit breakers** mandated by regulation — **Limit Up-Limit Down (LULD)** is the US equity market circuit breaker rule: if a stock's price moves more than a defined percentage in a short window, trading is paused for 15 minutes to prevent cascading crashes
- **Pre- and post-trade transparency** — publish order book data in real time

None of these are implemented in EduMatcher. They collectively represent years of engineering work for a real exchange.

3.19.11 Summary Table

Category	EduMatcher	Production
Message delivery	Sequenced (gaps detectable), no replay of missed messages	Sequence numbers, replay, guaranteed delivery
Authentication	Gateway ID only, no cryptographic proof	TLS certificates, challenge-response
Authorisation	Role-based (TRADER/MARKET_MAKER/ADMIN) — no enforcement of symbol restrictions	Fine-grained per-operation, per-symbol permissions
Failover	None	Primary/secondary with automatic failover
Clock precision	Integer nanoseconds via `now_ns()`	+ PTP synchronisation, hardware wire timestamps
Price arithmetic	Integer tick-based, configurable per symbol	+ Dynamic tick size changes, exchange tick tables
Order protocol	Custom text (`NEW|SYM=AAPL|...`)	FIX 4.4 / 5.0
Rate limiting	None	Per-gateway limits enforced at gateway
Pre-trade risk	Price collars, circuit breakers, kill switch	+ Per-gateway limits, fat finger thresholds
Regulatory reporting	None	Real-time trade reporting, audit retention
Trade persistence	SQLite (`clearing.db`)	Durable, replicated, queryable store
Replay buffer	None on the core bus (WS trades feed has a bounded in-memory resume)	Persistent log with configurable retention

= implemented in EduMatcher

3.20 Conclusion

EduMatcher implements the intellectual core of a matching engine — the part that is genuinely difficult to reason about — with real optimisations and real trade-offs, in clean Python that exposes the underlying decisions.

What it teaches:

- **Price-time priority** and how a heap encodes it in a single tuple key
- **Lazy deletion** as the right solution to the "no remove in a heap" problem
- **Single-writer architecture** as the simplest correct path to thread safety
- **Publish-subscribe messaging** as the decoupling pattern that lets each component evolve independently
- **The full order type taxonomy** and why each type exists
- **Auction mechanics** and how equilibrium price maximises matched volume

- **Integer tick arithmetic** and why float prices are insufficient for matching
- **The real gap** between educational and production systems

The codebase is designed to be read. Refer to the reading guide at the top of this document for the recommended sequence.

CHAPTER 4

SQLite Database Formats

EduMatcher uses SQLite for durable event records, derived summaries, and operator-interface state. Database paths normally live under `$DATA_DIR`, but services and bridges can override their individual paths. Each database has one owning writer and its own retention and recovery policy.

The complete database inventory, field metadata, and formal DDL are in the generated SQLite Schema Reference. It is built by running each owner's schema initializer against a temporary database; the tables, columns, indexes, and views are not duplicated here.

4.1 Storage Behavior

The statistics store retains daily rollups. Raw event and snapshot retention is configurable through `pm-stats`; the default keeps all rows. Clearing trade events default to a 90-day window, while current positions and daily summaries remain available beyond that window. The log server defaults to 30 days of log events; `log_server.retention_days: null` or zero keeps them indefinitely.

The market-simulation news store is not pruned automatically. If an older `market_sim_news.jsonl` exists, it is imported when the SQLite database is first created, and the source file remains untouched. Audit indexes are derived from retained audit logs and can be rebuilt.

The audit trail is line-oriented text, `[ts] [topic] [meta] {json}`, in three kinds of file under the data directory: `audit.log`, the active file holding one trading day (midnight in `locale.session_timezone`, by each line's receive time); `audit/audit-YYYY-MM-DD.N.log`, closed parts of a day, numbered in write order and never renamed; and `audit/audit-YYYY-MM-DD.log.zst`, one zstd stream per archived day, the day's parts concatenated, so decompressing it yields the day's log byte for byte. The archive's frame header records the uncompressed size. Readers take the live trail (unarchived parts, then `audit.log`) by default and a day's archive by `--date` or by path.

Trading prices and notionals in the statistics and clearing stores use integer ticks and require the row's `tick_decimals` to convert to display currency. Index levels and some derived values are `REAL` values. Timestamps may be UTC ISO-8601 text or integer Unix nanoseconds; the generated reference shows each column's declared type, and the owning process defines its units.

4.2 SQLite Files on Disk

`stats.db.lock` is a SQLite sidecar used only to hold the `pm-stats` writer lock; it contains no application tables. While a database is open in WAL mode, SQLite may also create `-wal` and `-shm` files. Keep these beside the database while its owner is running and do not remove them independently.

For a consistent backup, stop the database owner or use SQLite's backup API. Copying only the main database file while WAL writes are active can omit committed transactions. Confirm the effective path before inspecting or backing up a file.

4.3 Keeping the Schema Reference Current

The schema initializer used by a runtime owner is authoritative. After changing one, regenerate the reference and verify it:

```
make dbdocs
make dbdocs-check
```

`make check` and CI run the drift check. The generated chapter is not edited by hand; update the schema owner and regenerate instead.

CHAPTER 5

SQLite Schema Reference

This reference is generated by applying each database owner's schema initializer to a temporary SQLite database. Column tables and SQL below come from SQLite's `table_info` and `sqlite_schema`; DDL is not copied into this document by hand. Run `make dbdocs-check` to detect drift.

SQLite affinity types are shown as declared. The full DDL contains constraints, keys, defaults, indexes, and views where defined.

5.1 `stats.db`

Owner: pm-stats
Purpose: Market, order, and index statistics; raw event retention is configured by pm-stats.
Schema source: `src/edumatcher/stats/main.py`

5.1.1 Fields

Object	Field	Declared type	Not null	Default	Primary key
table: daily_stats	date	TEXT	yes		1
table: daily_stats	symbol	TEXT	yes		2
table: daily_stats	open_price	INTEGER	no		
table: daily_stats	high_price	INTEGER	no		
table: daily_stats	low_price	INTEGER	no		
table: daily_stats	close_price	INTEGER	no		
table: daily_stats	open_bid	INTEGER	no		
table: daily_stats	open_ask	INTEGER	no		
table: daily_stats	close_bid	INTEGER	no		
table: daily_stats	close_ask	INTEGER	no		

Object	Field	Declared type	Not null	Default	Primary key
table: daily_stats	volume	INTEGER	yes	0	
table: daily_stats	trade_count	INTEGER	yes	0	
table: daily_stats	turnover	INTEGER	yes	0	
table: daily_stats	vwap	REAL	no		
table: daily_stats	largest_trade_qty	INTEGER	no		
table: daily_stats	largest_trade_price	INTEGER	no		
table: daily_stats	tick_decimals	INTEGER	yes	2	
table: feed_gaps	seq	INTEGER	no		1
table: feed_gaps	ts	TEXT	yes		
table: feed_gaps	stream	TEXT	yes		
table: feed_gaps	expected_id	INTEGER	yes		
table: feed_gaps	received_id	INTEGER	yes		
table: feed_gaps	missing_count	INTEGER	yes		
table: index_daily_stats	date	TEXT	yes		1
table: index_daily_stats	index_id	TEXT	yes		2
table: index_daily_stats	open_level	REAL	no		
table: index_daily_stats	high_level	REAL	no		
table: index_daily_stats	low_level	REAL	no		
table: index_daily_stats	close_level	REAL	no		
table: index_daily_stats	close_session_state	TEXT	no		
table: index_daily_stats	open_aggregate_cap	REAL	no		
table: index_daily_stats	close_aggregate_cap	REAL	no		
table: index_daily_stats	update_count	INTEGER	yes	0	
table: index_level_snapshots	seq	INTEGER	no		1
table: index_level_snapshots	ts	TEXT	yes		
table: index_level_snapshots	index_id	TEXT	yes		

Object	Field	Declared type	Not null	Default	Primary key
table: index_level_snapshots	level	REAL	yes		
table: index_level_snapshots	aggregate_cap	REAL	no		
table: index_level_snapshots	divisor	REAL	no		
table: index_level_snapshots	session_state	TEXT	no		
table: index_level_snapshots	day_open	REAL	no		
table: index_level_snapshots	day_high	REAL	no		
table: index_level_snapshots	day_low	REAL	no		
table: instruments	symbol	TEXT	yes		1
table: instruments	tick_decimals	INTEGER	yes		
table: instruments	tick_size	REAL	yes		
table: instruments	currency	TEXT	no		
table: instruments	source	TEXT	yes		
table: instruments	updated_ts	TEXT	yes		
table: order_events	seq	INTEGER	no		1
table: order_events	ts	TEXT	yes		
table: order_events	event_type	TEXT	yes		
table: order_events	order_id	TEXT	yes		
table: order_events	gateway_id	TEXT	yes		
table: order_events	symbol	TEXT	yes		
table: order_events	side	TEXT	no		
table: order_events	order_type	TEXT	no		
table: order_events	tif	TEXT	no		
table: order_events	price	REAL	no		

Object	Field	Declared type	Not null	Default	Primary key
table: order_events	quantity	INTEGER	no		
table: order_events	remaining_qty	INTEGER	no		
table: order_events	status	TEXT	no		
table: order_events	fill_price	REAL	no		
table: order_events	fill_qty	INTEGER	no		
table: order_events	trade_ids	TEXT	no		
table: order_events	reason	TEXT	no		
table: order_events	reject_code	TEXT	no		
table: order_events	client_tag	TEXT	no		
table: order_events	combo_parent_id	TEXT	no		
table: order_events	oco_group_id	TEXT	no		
table: order_events	priority_reset	INTEGER	no		
table: order_events	msg_id	TEXT	no		
table: price_snapshots	ts	TEXT	yes		1
table: price_snapshots	symbol	TEXT	yes		2
table: price_snapshots	mid_price	REAL	no		
table: price_snapshots	best_bid	INTEGER	no		
table: price_snapshots	best_ask	INTEGER	no		
table: price_snapshots	pct_change	REAL	no		
table: price_snapshots	tick_decimals	INTEGER	yes	2	
table: stats_meta	key	TEXT	yes		1
table: stats_meta	value	TEXT	no		
table: trade_log	ts	TEXT	yes		
table: trade_log	trade_id	TEXT	yes		1
table: trade_log	symbol	TEXT	yes		
table: trade_log	price	INTEGER	yes		

Object	Field	Declared type	Not null	Default	Primary key
table: trade_log	quantity	INTEGER	yes		
table: trade_log	tick_decimals	INTEGER	yes	2	
table: trade_log	buy_gateway_id	TEXT	no		
table: trade_log	sell_gateway_id	TEXT	no		
table: trade_log	aggressor_side	TEXT	no		
table: trade_log	buy_order_id	TEXT	no		
table: trade_log	sell_order_id	TEXT	no		
table: trade_log	match_type	TEXT	no		
table: trade_log	msg_id	TEXT	no		

5.1.2 DDL

table daily_stats

```
CREATE TABLE daily_stats (
    date                TEXT NOT NULL,
    symbol              TEXT NOT NULL,
    open_price          INTEGER,
    high_price          INTEGER,
    low_price           INTEGER,
    close_price         INTEGER,
    open_bid            INTEGER,
    open_ask            INTEGER,
    close_bid           INTEGER,
    close_ask           INTEGER,
    volume              INTEGER NOT NULL DEFAULT 0,
    trade_count         INTEGER NOT NULL DEFAULT 0,
    -- Traded notional in ticks x quantity: sum(price_ticks * quantity). Exact,
    -- and the VWAP numerator — so vwap is exactly reproducible as
    -- turnover / volume without any float round-trip.
    turnover            INTEGER NOT NULL DEFAULT 0,
    -- Derived, therefore REAL: turnover / volume is a ratio and lands between
    -- ticks far more often than on one. The exact inputs are stored alongside,
    -- so a consumer needing full precision computes it rather than reading it.
    vwap                REAL,
    largest_trade_qty   INTEGER,
    largest_trade_price INTEGER,
    tick_decimals       INTEGER NOT NULL DEFAULT 2,
    PRIMARY KEY (date, symbol)
)
```

table feed_gaps

```
CREATE TABLE feed_gaps (
    seq             INTEGER PRIMARY KEY AUTOINCREMENT,
    ts              TEXT    NOT NULL,
    stream          TEXT    NOT NULL,
    -- The id we expected next, the one that actually arrived, and how many
    -- are unaccounted for between them.
    expected_id     INTEGER NOT NULL,
    received_id     INTEGER NOT NULL,
    missing_count   INTEGER NOT NULL
)
```

table index_daily_stats

```
CREATE TABLE index_daily_stats (
    date               TEXT NOT NULL,
    index_id           TEXT NOT NULL,
    open_level         REAL,
    high_level         REAL,
    low_level          REAL,
    close_level        REAL,
    -- session_state as of the most recent update, i.e. the one that set
    -- close_level. Only means "close_level is final" when this is CLOSED;
    -- for the current trading day it will typically show CONTINUOUS or
    -- another intraday state until pm-index's forced EOD publish arrives.
    close_session_state TEXT,
    open_aggregate_cap  REAL,
    close_aggregate_cap REAL,
    update_count       INTEGER NOT NULL DEFAULT 0,
    PRIMARY KEY (date, index_id)
)
```

table index_level_snapshots

```
CREATE TABLE index_level_snapshots (
    -- Arrival order. ts has millisecond resolution, so it cannot identify an
    -- update: two in one millisecond are two rows (K9 / S4).
    seq             INTEGER PRIMARY KEY AUTOINCREMENT,
    ts              TEXT NOT NULL,
    index_id        TEXT NOT NULL,
    level           REAL NOT NULL,
    aggregate_cap   REAL,
    divisor         REAL,
    session_state   TEXT,
    day_open        REAL,
    day_high        REAL,
    day_low         REAL
)
```

table instruments

```
CREATE TABLE instruments (
    symbol        TEXT    NOT NULL PRIMARY KEY,
    -- Decimal places a price trades in; 1 tick = 10^-tick_decimals.
    tick_decimals INTEGER NOT NULL,
    -- Stored as well as derivable, so a BI tool can join without arithmetic.
    tick_size     REAL    NOT NULL,
    -- RESERVED FOR FUTURE USE. EduMatcher has no currency model: prices carry
    -- no currency, aggregate_cap sums across whatever the constituents are
    -- priced in, and nothing converts. Always NULL today. The column exists so
    -- that adding a currency model later does not require a schema migration
    -- of the reference table, and so a consumer can see the field is absent
    -- rather than assume a currency.
    currency      TEXT,
    -- 'config'  — from the engine's symbol_meta, authoritative
    -- 'observed'— inferred from a message's tick_decimals, best effort
    source        TEXT    NOT NULL,
    updated_ts    TEXT    NOT NULL
)
```

table order_events

```
CREATE TABLE order_events (
    seq             INTEGER PRIMARY KEY AUTOINCREMENT,
    ts              TEXT NOT NULL,
    event_type      TEXT NOT NULL,
    order_id        TEXT NOT NULL,
    gateway_id      TEXT NOT NULL,
    symbol          TEXT NOT NULL,
    side            TEXT,
    order_type      TEXT,
    tif             TEXT,
    price           REAL,
    quantity        INTEGER,
    remaining_qty   INTEGER,
    status          TEXT,
    fill_price      REAL,
    fill_qty        INTEGER,
    -- JSON array of the durable engine trade ids behind a FILL. One fill
    -- event aggregates everything the order executed in one dispatch, so it
    -- can name several trades; NULL for every other event type.
    trade_ids       TEXT,
    reason          TEXT,
    -- The ack's reject_code on a *_REJECT event; NULL otherwise.
    reject_code     TEXT,
    client_tag      TEXT,
    combo_parent_id TEXT,
    oco_group_id    TEXT,
    priority_reset  INTEGER,
    -- The audit message id this row records (its line's msg=).
    msg_id          TEXT
)
```

table price_snapshots

```
CREATE TABLE price_snapshots (
    ts            TEXT NOT NULL,
    symbol        TEXT NOT NULL,
    -- Derived, therefore REAL and expressed in ticks: the midpoint of two
    -- adjacent ticks is a half tick, which no integer can hold. best_bid and
    -- best_ask beside it are exact.
    mid_price     REAL,
    best_bid      INTEGER,
    best_ask      INTEGER,
    -- A percentage, not money — never converted.
    pct_change    REAL,
    tick_decimals INTEGER NOT NULL DEFAULT 2,
    PRIMARY KEY (ts, symbol)
)
```

table stats_meta

```
CREATE TABLE stats_meta (
    key   TEXT NOT NULL PRIMARY KEY,
    value TEXT
)
```

table trade_log

```
CREATE TABLE trade_log (
    -- The durable engine trade id is the sole execution identity.
    ts              TEXT NOT NULL,
    trade_id        TEXT NOT NULL,
    symbol          TEXT NOT NULL,
    price           INTEGER NOT NULL,
    quantity        INTEGER NOT NULL,
    tick_decimals   INTEGER NOT NULL DEFAULT 2,
    buy_gateway_id  TEXT,
    sell_gateway_id TEXT,
    -- Mirrored from the engine payload verbatim: BUY or SELL for a
    -- continuous match, AUCTION for an uncross print where both sides were
    -- resting and there is no true aggressor. Without this column the table
    -- cannot support trade classification or order-flow imbalance, and
    -- auction prints are indistinguishable from continuous ones.
    aggressor_side  TEXT,
    buy_order_id    TEXT,
    sell_order_id   TEXT,
    -- How the trade came about (trade.executed.match_type).
    match_type      TEXT,
    -- The audit message id of the print (its line's msg=).
    msg_id          TEXT,
    PRIMARY KEY (trade_id)
)
```

index idx_ds_index_id_date

```
CREATE INDEX idx_ds_index_id_date ON index_daily_stats(index_id, date)
```

index idx_fg_ts

```
CREATE INDEX idx_fg_ts ON feed_gaps(ts)
```

index idx_ids_index_ts

```
CREATE INDEX idx_ids_index_ts ON index_level_snapshots(index_id, ts)
```

index idx_oe_gateway_ts

```
CREATE INDEX idx_oe_gateway_ts ON order_events(gateway_id, ts)
```

index idx_oe_order_id

```
CREATE INDEX idx_oe_order_id ON order_events(order_id)
```

index idx_oe_symbol_ts

```
CREATE INDEX idx_oe_symbol_ts ON order_events(symbol, ts)
```

index idx_oe_type_ts

```
CREATE INDEX idx_oe_type_ts ON order_events(event_type, ts)
```

index idx_ps_symbol_ts

```
CREATE INDEX idx_ps_symbol_ts ON price_snapshots(symbol, ts)
```

index idx_tl_symbol_qty

```
CREATE INDEX idx_tl_symbol_qty ON trade_log(symbol, quantity DESC, ts)
```

index idx_tl_symbol_ts

```
CREATE INDEX idx_tl_symbol_ts ON trade_log(symbol, ts)
```

5.2 clearing.db

Owner: pm-clearing
Purpose: Executions, gateway positions, daily summaries, and clearing metadata.
Schema source: `src/edumatcher/clearing/store.py`

5.2.1 Fields

Object	Field	Declared type	Not null	Default	Primary key
table: clearing_meta	key	TEXT	no		1
table: clearing_meta	value	TEXT	yes		
table: gateway_daily_summary	trade_date	TEXT	yes		1
table: gateway_daily_summary	gateway_id	TEXT	yes		2
table: gateway_daily_summary	symbol	TEXT	yes		3
table: gateway_daily_summary	traded_qty	INTEGER	yes		
table: gateway_daily_summary	traded_notional	INTEGER	yes		
table: gateway_daily_summary	buy_qty	INTEGER	yes		
table: gateway_daily_summary	sell_qty	INTEGER	yes		
table: gateway_daily_summary	buy_notional	INTEGER	yes		
table: gateway_daily_summary	sell_notional	INTEGER	yes		
table: gateway_daily_summary	net_amount	INTEGER	yes		
table: gateway_daily_summary	realized_pnl	REAL	yes		
table: gateway_daily_summary	end_net_qty	INTEGER	yes		
table: gateway_daily_summary	end_avg_cost	REAL	yes		
table: gateway_daily_summary	end_unrealized_pnl	REAL	yes		
table: gateway_daily_summary	tick_decimals	INTEGER	yes	2	
table: gateway_daily_summary	last_trade_ts_ns	INTEGER	no		

Object	Field	Declared type	Not null	Default	Primary key
table: gateway_daily_summary	updated_ts_ns	INTEGER	yes		
table: gateway_sessions	gateway_id	TEXT	yes		1
table: gateway_sessions	connected_at_ns	INTEGER	yes		2
table: gateway_sessions	disconnected_at_ns	INTEGER	no		
table: gateway_sessions	disconnect_reason	TEXT	no		
table: gateway_symbol_positions	gateway_id	TEXT	yes		1
table: gateway_symbol_positions	symbol	TEXT	yes		2
table: gateway_symbol_positions	net_qty	INTEGER	yes		
table: gateway_symbol_positions	avg_cost	REAL	yes		
table: gateway_symbol_positions	realized_pnl	REAL	yes		
table: gateway_symbol_positions	unrealized_pnl	REAL	yes		
table: gateway_symbol_positions	mark_price	INTEGER	no		
table: gateway_symbol_positions	tick_decimals	INTEGER	yes	2	
table: gateway_symbol_positions	buy_qty	INTEGER	yes		
table: gateway_symbol_positions	sell_qty	INTEGER	yes		
table: gateway_symbol_positions	buy_notional	INTEGER	yes		
table: gateway_symbol_positions	sell_notional	INTEGER	yes		
table: gateway_symbol_positions	last_trade_ts_ns	INTEGER	no		
table: gateway_symbol_positions	updated_ts_ns	INTEGER	yes		
table: position_close_snapshot	trade_date	TEXT	yes		1
table: position_close_snapshot	gateway_id	TEXT	yes		2
table: position_close_snapshot	symbol	TEXT	yes		3
table: position_close_snapshot	net_qty	INTEGER	yes		

Object	Field	Declared type	Not null	Default	Primary key
table: position_close_snapshot	realized_pnl	REAL	yes		
table: position_close_snapshot	unrealized_pnl	REAL	yes		
table: position_close_snapshot	mark_price	INTEGER	no		
table: position_close_snapshot	tick_decimals	INTEGER	yes	2	
table: session_events	id	INTEGER	no		1
table: session_events	event_type	TEXT	yes		
table: session_events	ts_ns	INTEGER	yes		
table: session_events	trade_date	TEXT	yes		
table: session_events	payload_json	TEXT	no		
table: trade_events	id	TEXT	yes		1
table: trade_events	ts_ns	INTEGER	yes		
table: trade_events	trade_date	TEXT	yes		
table: trade_events	symbol	TEXT	yes		
table: trade_events	quantity	INTEGER	yes		
table: trade_events	price	INTEGER	yes		
table: trade_events	tick_decimals	INTEGER	yes	2	
table: trade_events	buy_order_id	TEXT	no		
table: trade_events	sell_order_id	TEXT	no		
table: trade_events	buy_gateway_id	TEXT	yes		
table: trade_events	sell_gateway_id	TEXT	yes		
table: trade_events	aggressor_side	TEXT	no		
table: trade_events	ingest_ts_ns	INTEGER	yes		
table: trade_events	match_type	TEXT	no		

Object	Field	Declared type	Not null	Default	Primary key
table: trade_events	msg_id	TEXT	no		
view: daily_exchange_totals	trade_date	TEXT	no		
view: daily_exchange_totals	traded_qty_total	BLOB	no		
view: daily_exchange_totals	traded_notional_total	BLOB	no		
view: daily_exchange_totals	net_amount_total	BLOB	no		
view: daily_exchange_totals	realized_pnl_total	BLOB	no		
view: gateway_pnl_totals	gateway_id	TEXT	no		
view: gateway_pnl_totals	realized_pnl_total	BLOB	no		
view: gateway_pnl_totals	unrealized_pnl_total	BLOB	no		
view: gateway_pnl_totals	total_pnl	BLOB	no		
view: gateway_pnl_totals	net_qty_total	BLOB	no		

5.2.2 DDL

table clearing_meta

```
CREATE TABLE clearing_meta (
  key   TEXT PRIMARY KEY,
  value TEXT NOT NULL
)
```

table gateway_daily_summary

```
CREATE TABLE gateway_daily_summary (
  trade_date         TEXT    NOT NULL,
  gateway_id         TEXT    NOT NULL,
  symbol             TEXT    NOT NULL,
  traded_qty         INTEGER NOT NULL,
  traded_notional    INTEGER NOT NULL,
  buy_qty            INTEGER NOT NULL,
  sell_qty           INTEGER NOT NULL,
  buy_notional       INTEGER NOT NULL,
  sell_notional      INTEGER NOT NULL,
  net_amount         INTEGER NOT NULL,
  realized_pnl       REAL    NOT NULL,
  end_net_qty        INTEGER NOT NULL,
  end_avg_cost       REAL    NOT NULL,
  end_unrealized_pnl REAL    NOT NULL,
    tick_decimals      INTEGER NOT NULL DEFAULT 2,
  last_trade_ts_ns   INTEGER,
  updated_ts_ns      INTEGER NOT NULL,
  PRIMARY KEY (trade_date, gateway_id, symbol)
)
```

table gateway_sessions

```
CREATE TABLE gateway_sessions (
  gateway_id        TEXT    NOT NULL,
  connected_at_ns   INTEGER NOT NULL,
  disconnected_at_ns INTEGER,
  disconnect_reason TEXT,
  PRIMARY KEY (gateway_id, connected_at_ns)
)
```

table gateway_symbol_positions

```
CREATE TABLE gateway_symbol_positions (
  gateway_id       TEXT    NOT NULL,
  symbol           TEXT    NOT NULL,
  net_qty          INTEGER NOT NULL,
  avg_cost         REAL    NOT NULL,
  realized_pnl     REAL    NOT NULL,
  unrealized_pnl   REAL    NOT NULL,
  mark_price       INTEGER,
    tick_decimals    INTEGER NOT NULL DEFAULT 2,
  buy_qty          INTEGER NOT NULL,
  sell_qty         INTEGER NOT NULL,
  buy_notional     INTEGER NOT NULL,
  sell_notional    INTEGER NOT NULL,
  last_trade_ts_ns INTEGER,
  updated_ts_ns    INTEGER NOT NULL,
  PRIMARY KEY (gateway_id, symbol)
)
```

table position_close_snapshot

```
CREATE TABLE position_close_snapshot (
  trade_date     TEXT    NOT NULL,
  gateway_id     TEXT    NOT NULL,
  symbol         TEXT    NOT NULL,
  net_qty        INTEGER NOT NULL,
  realized_pnl   REAL    NOT NULL,
  unrealized_pnl REAL    NOT NULL,
  mark_price     INTEGER,
  tick_decimals  INTEGER NOT NULL DEFAULT 2,
  PRIMARY KEY (trade_date, gateway_id, symbol)
)
```

table session_events

```
CREATE TABLE session_events (
  id           INTEGER PRIMARY KEY AUTOINCREMENT,
  event_type   TEXT    NOT NULL,
  ts_ns        INTEGER NOT NULL,
  trade_date   TEXT    NOT NULL,
  payload_json TEXT
)
```

table trade_events

```
CREATE TABLE trade_events (
  -- The engine's durable trade id is the sole execution identity.
  id               TEXT    NOT NULL,
  ts_ns            INTEGER NOT NULL,
  trade_date       TEXT    NOT NULL,
  symbol           TEXT    NOT NULL,
  quantity         INTEGER NOT NULL,
  price            INTEGER NOT NULL,
    tick_decimals    INTEGER NOT NULL DEFAULT 2,
  buy_order_id     TEXT,
  sell_order_id    TEXT,
  buy_gateway_id   TEXT    NOT NULL,
  sell_gateway_id  TEXT    NOT NULL,
  -- Mirrored from the engine payload verbatim.  See module docstring note:
  -- auction trades can carry a biased aggressor marker upstream.
  aggressor_side   TEXT,
  ingest_ts_ns     INTEGER NOT NULL,
  -- How the trade came about (trade.executed.match_type), and the audit's
  -- message id of the print: joins a row to its audit line.
  match_type       TEXT,
  msg_id           TEXT,
  PRIMARY KEY (id)
)
```

view daily_exchange_totals

```
CREATE VIEW daily_exchange_totals AS
SELECT
  trade_date,
  SUM(traded_qty)               AS traded_qty_total,
  SUM(traded_notional / scale)  AS traded_notional_total,
  SUM(net_amount / scale)       AS net_amount_total,
  SUM(realized_pnl / scale)     AS realized_pnl_total
FROM (
  SELECT
    trade_date, traded_qty, traded_notional, net_amount, realized_pnl,
    CASE tick_decimals
      WHEN 0 THEN 1.0        WHEN 1 THEN 10.0
      WHEN 2 THEN 100.0      WHEN 3 THEN 1000.0
      WHEN 4 THEN 10000.0    WHEN 5 THEN 100000.0
      WHEN 6 THEN 1000000.0  WHEN 7 THEN 10000000.0
      WHEN 8 THEN 100000000.0 ELSE 100.0
    END AS scale
  FROM gateway_daily_summary
)
GROUP BY trade_date
```

view gateway_pnl_totals

```
CREATE VIEW gateway_pnl_totals AS
SELECT
  gateway_id,
  SUM(realized_pnl / scale)                    AS realized_pnl_total,
  SUM(unrealized_pnl / scale)                  AS unrealized_pnl_total,
  SUM((realized_pnl + unrealized_pnl) / scale) AS total_pnl,
  SUM(net_qty)                                 AS net_qty_total
FROM (
  SELECT
    gateway_id, net_qty, realized_pnl, unrealized_pnl,
    CASE tick_decimals
      WHEN 0 THEN 1.0        WHEN 1 THEN 10.0
      WHEN 2 THEN 100.0      WHEN 3 THEN 1000.0
      WHEN 4 THEN 10000.0    WHEN 5 THEN 100000.0
      WHEN 6 THEN 1000000.0  WHEN 7 THEN 10000000.0
      WHEN 8 THEN 100000000.0 ELSE 100.0
    END AS scale
  FROM gateway_symbol_positions
)
GROUP BY gateway_id
```

index ix_gateway_sessions_gateway

```
CREATE INDEX ix_gateway_sessions_gateway
  ON gateway_sessions(gateway_id)
```

index ix_gds_gateway_date

```
CREATE INDEX ix_gds_gateway_date
  ON gateway_daily_summary(gateway_id, trade_date)
```

index ix_gds_symbol_date

```
CREATE INDEX ix_gds_symbol_date
  ON gateway_daily_summary(symbol, trade_date)
```

index ix_gsp_gateway

```
CREATE INDEX ix_gsp_gateway
  ON gateway_symbol_positions(gateway_id)
```

index ix_gsp_symbol

```
CREATE INDEX ix_gsp_symbol
  ON gateway_symbol_positions(symbol)
```

index ix_session_events_date

```
CREATE INDEX ix_session_events_date
  ON session_events(trade_date)
```

index ix_session_events_type

```
CREATE INDEX ix_session_events_type
  ON session_events(event_type)
```

index ix_trade_events_buy_gw_date

```
CREATE INDEX ix_trade_events_buy_gw_date
  ON trade_events(buy_gateway_id, trade_date)
```

index ix_trade_events_date

```
CREATE INDEX ix_trade_events_date
  ON trade_events(trade_date)
```

index ix_trade_events_sell_gw_date

```
CREATE INDEX ix_trade_events_sell_gw_date
  ON trade_events(sell_gateway_id, trade_date)
```

index ix_trade_events_symbol_date

```
CREATE INDEX ix_trade_events_symbol_date
  ON trade_events(symbol, trade_date)
```

5.3 eod.db

Owner: pm-eod
Purpose: End-of-day settlement ledger: each trader's amount per trading date, in integer minor units (1234 = 12.34) of the display currency; positive = the exchange owes the trader, negative = the trader owes the exchange; every date sums to zero.
Schema source: `src/edumatcher/eod/ledger.py`

5.3.1 Fields

Object	Field	Declared type	Not null	Default	Primary key
table: eod_runs	trade_date	TEXT	no		1
table: eod_runs	status	TEXT	yes		
table: eod_runs	started_at	TEXT	yes		
table: eod_runs	finished_at	TEXT	yes		
table: eod_runs	steps_json	TEXT	yes		
table: eod_settlement	trade_date	TEXT	yes		1
table: eod_settlement	account	TEXT	yes		2
table: eod_settlement	amount_minor	INTEGER	yes		
table: eod_settlement	detail_json	TEXT	yes		

5.3.2 DDL

table eod_runs

```
CREATE TABLE eod_runs (
  trade_date  TEXT PRIMARY KEY,
  status      TEXT NOT NULL,   -- ok | failed | report-failed
  started_at  TEXT NOT NULL,   -- ISO 8601, UTC
  finished_at TEXT NOT NULL,   -- ISO 8601, UTC
  steps_json  TEXT NOT NULL    -- [{"step", "status", "message"}, ...]
)
```

table eod_settlement

```
CREATE TABLE eod_settlement (
  trade_date   TEXT    NOT NULL,  -- YYYY-MM-DD, in the session timezone
  -- A gateway id, or 'EXCHANGE' for the rounding residual.
  account      TEXT    NOT NULL,
  -- The day's change in the account's mark-to-market equity, in integer minor
  -- units of the display currency (2 decimals: 1234 = 12.34).  Positive: the
  -- exchange owes the trader (credit).  Negative: the trader owes the
  -- exchange (debit).  A running balance is the sum of these plus the
  -- roster's opening balance; it is never stored.
  amount_minor INTEGER NOT NULL,
  -- {symbol: day P&L as a decimal string in display currency, same sign as
  -- amount_minor}; the unrounded parts the statement lists per symbol.
  detail_json  TEXT    NOT NULL,
  PRIMARY KEY (trade_date, account)
)
```

5.4 log.db

Owner: pm-log-srv
Purpose: Process log events, connection sessions, and server counters.
Schema source: `src/edumatcher/log_srv/schema.py`

5.4.1 Fields

Object	Field	Declared type	Not null	Default	Primary key
table: log_events	seq	INTEGER	no		1
table: log_events	client_ts	TEXT	yes		
table: log_events	server_ts	TEXT	yes		
table: log_events	process	TEXT	yes		
table: log_events	instance	TEXT	no		
table: log_events	pid	INTEGER	yes		
table: log_events	host	TEXT	yes		
table: log_events	session	TEXT	yes		
table: log_events	level	TEXT	yes		
table: log_events	logger	TEXT	yes		
table: log_events	module	TEXT	no		
table: log_events	line	INTEGER	no		
table: log_events	has_exception	INTEGER	yes	0	
table: log_events	truncated	INTEGER	yes	0	
table: log_events	message	TEXT	yes		
table: processes	session	TEXT	no		1
table: processes	process	TEXT	yes		
table: processes	instance	TEXT	no		
table: processes	pid	INTEGER	yes		
table: processes	host	TEXT	yes		
table: processes	connected_at	TEXT	yes		
table: processes	last_seen_at	TEXT	yes		

Object	Field	Declared type	Not null	Default	Primary key
table: processes	last_heartbeat_at	TEXT	no		
table: processes	disconnected_at	TEXT	no		
table: processes	disconnect_reason	TEXT	no		
table: processes	log_count	INTEGER	yes	0	
table: server_stats	id	INTEGER	no		1
table: server_stats	started_at	TEXT	yes		
table: server_stats	total_log_events	INTEGER	yes	0	
table: server_stats	total_connections	INTEGER	yes	0	
table: server_stats	total_truncated	INTEGER	yes	0	
table: server_stats	total_errors_sent	INTEGER	yes	0	

5.4.2 DDL

table log_events

```
CREATE TABLE log_events (
    seq             INTEGER PRIMARY KEY AUTOINCREMENT,
    client_ts       TEXT NOT NULL,
    server_ts       TEXT NOT NULL,
    process         TEXT NOT NULL,
    instance        TEXT,
    pid             INTEGER NOT NULL,
    host            TEXT NOT NULL,
    session         TEXT NOT NULL,
    level           TEXT NOT NULL,
    logger          TEXT NOT NULL,
    module          TEXT,
    line            INTEGER,
    has_exception   INTEGER NOT NULL DEFAULT 0,
    truncated       INTEGER NOT NULL DEFAULT 0,
    message         TEXT NOT NULL
)
```

table processes

```
CREATE TABLE processes (
    session         TEXT PRIMARY KEY,
    process         TEXT NOT NULL,
    instance        TEXT,
    pid             INTEGER NOT NULL,
    host            TEXT NOT NULL,
    connected_at    TEXT NOT NULL,
    last_seen_at    TEXT NOT NULL,
    last_heartbeat_at TEXT,
    disconnected_at TEXT,
    disconnect_reason TEXT,
    log_count       INTEGER NOT NULL DEFAULT 0
)
```

table server_stats

```
CREATE TABLE server_stats (
    id                  INTEGER PRIMARY KEY CHECK (id = 1),
    started_at          TEXT NOT NULL,
    total_log_events    INTEGER NOT NULL DEFAULT 0,
    total_connections   INTEGER NOT NULL DEFAULT 0,
    total_truncated     INTEGER NOT NULL DEFAULT 0,
    total_errors_sent   INTEGER NOT NULL DEFAULT 0
)
```

index idx_le_client_ts

```
CREATE INDEX idx_le_client_ts  ON log_events(client_ts)
```

index idx_le_level_ts

```
CREATE INDEX idx_le_level_ts   ON log_events(level, client_ts)
```

index idx_le_logger_ts

```
CREATE INDEX idx_le_logger_ts  ON log_events(logger, client_ts)
```

index idx_le_process_ts

```
CREATE INDEX idx_le_process_ts ON log_events(process, client_ts)
```

index idx_le_session

```
CREATE INDEX idx_le_session    ON log_events(session)
```

index idx_proc_process

```
CREATE INDEX idx_proc_process ON processes(process, connected_at)
```

5.5 metrics.db

Owner: pm-monitor
Purpose: Operator-dashboard history: raw and 1-minute samples, daily totals, counter state and events.
Schema source: `src/edumatcher/monitor/store.py`

5.5.1 Fields

Object	Field	Declared type	Not null	Default	Primary key
table: counter_last	source	TEXT	yes		1
table: counter_last	name	TEXT	yes		2
table: counter_last	label	TEXT	yes	''	3
table: counter_last	boot_id	TEXT	yes		
table: counter_last	value	REAL	yes		
table: counter_last	ts	TEXT	yes		
table: daily_totals	date	TEXT	yes		1
table: daily_totals	source	TEXT	yes		4
table: daily_totals	name	TEXT	yes		2
table: daily_totals	label	TEXT	yes	''	3

Object	Field	Declared type	Not null	Default	Primary key
table: daily_totals	total	REAL	yes		
table: events	seq	INTEGER	no		1
table: events	ts	TEXT	yes		
table: events	kind	TEXT	yes		
table: events	severity	TEXT	yes		
table: events	subject	TEXT	yes		
table: events	detail	TEXT	yes	''	
table: metrics_meta	key	TEXT	no		1
table: metrics_meta	value	TEXT	yes		
table: samples_1m	series	INTEGER	yes		1
table: samples_1m	ts	TEXT	yes		2
table: samples_1m	v_min	REAL	no		
table: samples_1m	v_avg	REAL	no		
table: samples_1m	v_max	REAL	no		
table: samples_1m	v_last	REAL	no		
table: samples_1m	n	INTEGER	yes		
table: samples_1m	delta	REAL	no		
table: samples_raw	series	INTEGER	yes		1
table: samples_raw	ts	TEXT	yes		2
table: samples_raw	value	REAL	yes		
table: series	id	INTEGER	no		1
table: series	source	TEXT	yes		
table: series	name	TEXT	yes		
table: series	label	TEXT	yes	''	
view: v_samples_1m	ts	TEXT	no		
view: v_samples_1m	source	TEXT	no		
view: v_samples_1m	name	TEXT	no		
view: v_samples_1m	label	TEXT	no		
view: v_samples_1m	v_min	REAL	no		
view: v_samples_1m	v_avg	REAL	no		
view: v_samples_1m	v_max	REAL	no		
view: v_samples_1m	v_last	REAL	no		
view: v_samples_1m	n	INTEGER	no		
view: v_samples_1m	delta	REAL	no		
view: v_samples_raw	ts	TEXT	no		
view: v_samples_raw	source	TEXT	no		
view: v_samples_raw	name	TEXT	no		
view: v_samples_raw	label	TEXT	no		
view: v_samples_raw	value	REAL	no		

5.5.2 DDL

table counter_last

```
CREATE TABLE counter_last (
    source  TEXT NOT NULL,
    name    TEXT NOT NULL,
    label   TEXT NOT NULL DEFAULT '',
    boot_id TEXT NOT NULL,
    value   REAL NOT NULL,
    ts      TEXT NOT NULL,
    PRIMARY KEY (source, name, label)
)
```

table daily_totals

```
CREATE TABLE daily_totals (
    date    TEXT NOT NULL,          -- trading date in the session timezone
    source  TEXT NOT NULL,
    name    TEXT NOT NULL,
    label   TEXT NOT NULL DEFAULT '',
    total   REAL NOT NULL,
    PRIMARY KEY (date, name, label, source)
)
```

table events

```
CREATE TABLE events (
    seq      INTEGER PRIMARY KEY AUTOINCREMENT,  -- numbered by pm-monitor; AUTOINCREMENT keeps the high-water mark
    ts       TEXT NOT NULL,
    kind     TEXT NOT NULL,
    severity TEXT NOT NULL,
    subject  TEXT NOT NULL,         -- gateway id, process key, symbol, rule
    detail   TEXT NOT NULL DEFAULT ''
)
```

table metrics_meta

```
CREATE TABLE metrics_meta (
    key     TEXT PRIMARY KEY,
    value   TEXT NOT NULL
)
```

table samples_1m

```
CREATE TABLE samples_1m (
    series  INTEGER NOT NULL,       -- series.id
    ts      TEXT NOT NULL,          -- bucket start
    v_min   REAL, v_avg REAL, v_max REAL, v_last REAL,
    n       INTEGER NOT NULL,       -- samples in the bucket
    delta   REAL,                   -- counters: reset-corrected increment
    PRIMARY KEY (series, ts)
) WITHOUT ROWID
```

table samples_raw

```
CREATE TABLE samples_raw (
    series  INTEGER NOT NULL,       -- series.id
    ts      TEXT NOT NULL,
    value   REAL NOT NULL,
    PRIMARY KEY (series, ts)
) WITHOUT ROWID
```

table series

```
CREATE TABLE series (
    id      INTEGER PRIMARY KEY,
    source  TEXT NOT NULL,          -- "process/instance@host" or "host"
    name    TEXT NOT NULL,
    label   TEXT NOT NULL DEFAULT '',
    UNIQUE (name, label, source)
)
```

view v_samples_1m

```
CREATE VIEW v_samples_1m AS
    SELECT m.ts, s.source, s.name, s.label,
           m.v_min, m.v_avg, m.v_max, m.v_last, m.n, m.delta
    FROM samples_1m m JOIN series s ON s.id = m.series
```

view v_samples_raw

```
CREATE VIEW v_samples_raw AS
    SELECT r.ts, s.source, s.name, s.label, r.value
    FROM samples_raw r JOIN series s ON s.id = r.series
```

index idx_daily_name

```
CREATE INDEX idx_daily_name ON daily_totals(name, label, date)
```

index idx_ev_ts

```
CREATE INDEX idx_ev_ts ON events(ts)
```

5.6 market_sim_news.db

Owner: pm-market-sim
Purpose: Generated and instructor-published news events, and the value model's jumps.
Schema source: `src/edumatcher/market_sim/news_history.py`

5.6.1 Fields

Object	Field	Declared type	Not null	Default	Primary key
table: news_events	event_seq	INTEGER	no		1
table: news_events	id	TEXT	yes		
table: news_events	ts_ns	INTEGER	yes		
table: news_events	scope	TEXT	yes		
table: news_events	targets	TEXT	yes		
table: news_events	kind	TEXT	yes		
table: news_events	status	TEXT	yes		
table: news_events	headline	TEXT	yes		
table: news_events	sentiment	REAL	yes		
table: news_events	credibility	REAL	no		
table: news_events	related_id	TEXT	yes		
table: value_jumps	ts_ns	INTEGER	yes		
table: value_jumps	symbol	TEXT	yes		

Object	Field	Declared type	Not null	Default	Primary key
table: value_jumps	log_jump	REAL	yes		
table: value_jumps	value	REAL	yes		

5.6.2 DDL

table news_events

```
CREATE TABLE news_events (
            event_seq INTEGER PRIMARY KEY AUTOINCREMENT,
            id TEXT NOT NULL,
            ts_ns INTEGER NOT NULL,
            scope TEXT NOT NULL,
            targets TEXT NOT NULL,
            kind TEXT NOT NULL,
            status TEXT NOT NULL,
            headline TEXT NOT NULL,
            sentiment REAL NOT NULL,
            credibility REAL,
            related_id TEXT NOT NULL
        )
```

table value_jumps

```
CREATE TABLE value_jumps (
            ts_ns INTEGER NOT NULL,
            symbol TEXT NOT NULL,
            log_jump REAL NOT NULL,
            value REAL NOT NULL
        )
```

index news_events_status

```
CREATE INDEX news_events_status ON news_events(status)
```

index news_events_ts

```
CREATE INDEX news_events_ts ON news_events(ts_ns DESC)
```

index value_jumps_ts

```
CREATE INDEX value_jumps_ts ON value_jumps(ts_ns)
```

5.7 companies.db

Owner: pm-compdb-cli
Purpose: Made-up company profiles of the simulated market's tickers.
Schema source: `src/edumatcher/compdb/store.py`

5.7.1 Fields

Object	Field	Declared type	Not null	Default	Primary key
table: companies	symbol	TEXT	no		1
table: companies	name	TEXT	yes		
table: companies	sector	TEXT	yes		

Object	Field	Declared type	Not null	Default	Primary key
table: companies	industry	TEXT	yes		
table: companies	headquarters	TEXT	no		
table: companies	purpose	TEXT	no		
table: companies	key_market	TEXT	no		
table: companies	market_position	TEXT	no		
table: companies	origin	TEXT	no		
table: companies	founders	TEXT	no		
table: companies	founder_story	TEXT	no		
table: companies	founded_year	INTEGER	no		
table: companies	ipo_year	INTEGER	no		
table: companies	ipo_valuation_musd	REAL	no		
table: companies	ceo_name	TEXT	no		
table: companies	ceo_since	INTEGER	no		
table: companies	ceo_industry_years	INTEGER	no		
table: companies	employees	INTEGER	no		
table: companies	fiscal_year	INTEGER	no		
table: companies	revenue_musd	REAL	no		
table: companies	net_income_musd	REAL	no		
table: companies	revenue_growth_pct	REAL	no		
table: companies	operating_margin_pct	REAL	no		
table: companies	free_cash_flow_musd	REAL	no		
table: companies	market_cap_musd	REAL	no		
table: companies	dividend_yield_pct	REAL	no		
table: companies	debt_to_equity	REAL	no		
table: companies	credit_rating	TEXT	no		
table: companies	competitors	TEXT	no		
table: companies	key_risk	TEXT	no		
view: company_view	symbol	TEXT	no		
view: company_view	name	TEXT	no		
view: company_view	sector	TEXT	no		
view: company_view	industry	TEXT	no		
view: company_view	headquarters	TEXT	no		
view: company_view	purpose	TEXT	no		
view: company_view	key_market	TEXT	no		
view: company_view	market_position	TEXT	no		
view: company_view	origin	TEXT	no		
view: company_view	founders	TEXT	no		
view: company_view	founder_story	TEXT	no		
view: company_view	founded_year	INTEGER	no		
view: company_view	ipo_year	INTEGER	no		

Object	Field	Declared type	Not null	Default	Primary key
view: company_view	ipo_valuation_musd	REAL	no		
view: company_view	ceo_name	TEXT	no		
view: company_view	ceo_since	INTEGER	no		
view: company_view	ceo_industry_years	INTEGER	no		
view: company_view	employees	INTEGER	no		
view: company_view	fiscal_year	INTEGER	no		
view: company_view	revenue_musd	REAL	no		
view: company_view	net_income_musd	REAL	no		
view: company_view	revenue_growth_pct	REAL	no		
view: company_view	operating_margin_pct	REAL	no		
view: company_view	free_cash_flow_musd	REAL	no		
view: company_view	market_cap_musd	REAL	no		
view: company_view	dividend_yield_pct	REAL	no		
view: company_view	debt_to_equity	REAL	no		
view: company_view	credit_rating	TEXT	no		
view: company_view	competitors	TEXT	no		
view: company_view	key_risk	TEXT	no		
view: company_view	years_in_business	BLOB	no		
view: company_view	ceo_tenure_years	BLOB	no		
view: company_view	pe_ratio	BLOB	no		
view: company_view	ps_ratio	BLOB	no		
view: company_view	net_margin_pct	BLOB	no		

5.7.2 DDL

table companies

```
CREATE TABLE companies (
    symbol TEXT PRIMARY KEY,
    name TEXT NOT NULL,
    sector TEXT NOT NULL,
    industry TEXT NOT NULL,
    headquarters TEXT,
    purpose TEXT,
    key_market TEXT,
    market_position TEXT,
    origin TEXT,
    founders TEXT,
    founder_story TEXT,
    founded_year INTEGER,
    ipo_year INTEGER,
    ipo_valuation_musd REAL,
    ceo_name TEXT,
    ceo_since INTEGER,
    ceo_industry_years INTEGER,
    employees INTEGER,
    fiscal_year INTEGER,
    revenue_musd REAL,
    net_income_musd REAL,
    revenue_growth_pct REAL,
    operating_margin_pct REAL,
    free_cash_flow_musd REAL,
    market_cap_musd REAL,
    dividend_yield_pct REAL,
    debt_to_equity REAL,
    credit_rating TEXT,
    competitors TEXT,
    key_risk TEXT
)
```

view company_view

```
CREATE VIEW company_view AS
SELECT *,
    CAST(strftime('%Y', 'now') AS INTEGER) - founded_year AS years_in_business,
    CAST(strftime('%Y', 'now') AS INTEGER) - ceo_since AS ceo_tenure_years,
    CASE WHEN net_income_musd > 0 THEN ROUND(market_cap_musd / net_income_musd, 1) END AS pe_ratio,
    CASE WHEN revenue_musd > 0 THEN ROUND(market_cap_musd / revenue_musd, 2) END AS ps_ratio,
    CASE WHEN revenue_musd > 0 THEN ROUND(100.0 * net_income_musd / revenue_musd, 1) END AS net_margin_pct
FROM companies
```

index companies_sector

```
CREATE INDEX companies_sector ON companies(sector)
```

5.8 log-ui-acks.db

Owner: Log GUI bridge
Purpose: Operator issue acknowledgements and acknowledgement history.
Schema source: `web-apps/log-gui/apps/bridge/src/schema.sql`

5.8.1 Fields

Object	Field	Declared type	Not null	Default	Primary key
table: ack_history	id	INTEGER	no		1
table: ack_history	fingerprint	TEXT	yes		
table: ack_history	action	TEXT	yes		
table: ack_history	at	TEXT	yes		
table: ack_history	by	TEXT	yes		
table: ack_history	note	TEXT	no		
table: issue_acks	fingerprint	TEXT	no		1
table: issue_acks	acked_at	TEXT	yes		
table: issue_acks	acked_by	TEXT	yes		
table: issue_acks	note	TEXT	no		
table: issue_acks	acked_through_seq	INTEGER	yes		
table: issue_acks	level	TEXT	yes		
table: issue_acks	process	TEXT	yes		
table: issue_acks	logger	TEXT	yes		
table: issue_acks	sample_message	TEXT	yes		

5.8.2 DDL

table ack_history

```
CREATE TABLE ack_history (
    id           INTEGER PRIMARY KEY AUTOINCREMENT,
    fingerprint  TEXT NOT NULL,
    action       TEXT NOT NULL,
    at           TEXT NOT NULL,
    by            TEXT NOT NULL,
    note         TEXT
)
```

table issue_acks

```
CREATE TABLE issue_acks (
    fingerprint        TEXT PRIMARY KEY,
    acked_at           TEXT NOT NULL,
    acked_by           TEXT NOT NULL,
    note               TEXT,
    acked_through_seq  INTEGER NOT NULL,
    level              TEXT NOT NULL,
    process            TEXT NOT NULL,
    logger             TEXT NOT NULL,
    sample_message     TEXT NOT NULL
)
```

index idx_acks_at

```
CREATE INDEX idx_acks_at ON issue_acks(acked_at)
```

5.9 audit_index.db

Owner: pm-audit-cli
Purpose: Rebuildable query index over the rotating audit log.
Schema source: `src/edumatcher/audit/indexer.py`

5.9.1 Fields

Object	Field	Declared type	Not null	Default	Primary key
table: audit_events	id	INTEGER	no		1
table: audit_events	timestamp	TEXT	yes		
table: audit_events	topic	TEXT	yes		
table: audit_events	payload	TEXT	yes		
table: audit_events	gateway_id	TEXT	no		
table: audit_events	symbol	TEXT	no		
table: audit_events	order_id	TEXT	no		
table: audit_events	trade_id	TEXT	no		
table: audit_events	event_type	TEXT	no		
table: audit_events	seq	INTEGER	no		
table: audit_events	msg_id	TEXT	no		
table: audit_events	causation_id	TEXT	no		
table: audit_events	correlation_id	TEXT	no		
table: index_meta	key	TEXT	no		1
table: index_meta	value	TEXT	yes		

5.9.2 DDL

table audit_events

```
CREATE TABLE audit_events (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp   TEXT    NOT NULL,
    topic       TEXT    NOT NULL,
    payload     TEXT    NOT NULL,
    gateway_id  TEXT,
    symbol      TEXT,
    order_id    TEXT,
    trade_id    TEXT,
    event_type  TEXT,
    seq            INTEGER,
    msg_id         TEXT,
    causation_id   TEXT,
    correlation_id TEXT
)
```

table index_meta

```
CREATE TABLE index_meta (
    key   TEXT PRIMARY KEY,
    value TEXT NOT NULL
)
```

index idx_causation

```
CREATE INDEX idx_causation ON audit_events(causation_id)
```

index idx_chain

```
CREATE INDEX idx_chain     ON audit_events(correlation_id)
```

index idx_event_type

```
CREATE INDEX idx_event_type ON audit_events(event_type)
```

index idx_gateway

```
CREATE INDEX idx_gateway    ON audit_events(gateway_id)
```

index idx_gw_ts

```
CREATE INDEX idx_gw_ts      ON audit_events(gateway_id, timestamp)
```

index idx_msg_id

```
CREATE INDEX idx_msg_id    ON audit_events(msg_id)
```

index idx_order

```
CREATE INDEX idx_order      ON audit_events(order_id)
```

index idx_sym_ts

```
CREATE INDEX idx_sym_ts     ON audit_events(symbol, timestamp)
```

index idx_symbol

```
CREATE INDEX idx_symbol     ON audit_events(symbol)
```

index idx_timestamp

```
CREATE INDEX idx_timestamp  ON audit_events(timestamp)
```

index idx_topic

```
CREATE INDEX idx_topic      ON audit_events(topic)
```

index idx_topic_ts

```
CREATE INDEX idx_topic_ts   ON audit_events(topic, timestamp)
```

index idx_trade

```
CREATE INDEX idx_trade      ON audit_events(trade_id)
```

5.10 audit_replay.db

Owner: pm-audit-replay
Purpose: Rebuildable episode, event, causal-link, anomaly, and actor index.
Schema source: `src/edumatcher/audit/replay/index.py`

5.10.1 Fields

Object	Field	Declared type	Not null	Default	Primary key
table: actors	gateway_id	TEXT	no		1
table: actors	description	TEXT	no		
table: actors	first_seen	TEXT	yes		
table: actors	last_seen	TEXT	yes		

Object	Field	Declared type	Not null	Default	Primary key
table: actors	orders	INTEGER	yes	0	
table: actors	trades	INTEGER	yes	0	
table: anomalies	anomaly_id	INTEGER	no		1
table: anomalies	code	TEXT	yes		
table: anomalies	severity	TEXT	yes		
table: anomalies	episode_id	INTEGER	no		
table: anomalies	sort_key	TEXT	yes		
table: anomalies	receipt_ts	TEXT	yes		
table: anomalies	detail	TEXT	yes		
table: anomalies	file	TEXT	no		
table: anomalies	line_no	INTEGER	yes	0	
table: episode_events	episode_id	INTEGER	yes		1
table: episode_events	seq_in_ep	INTEGER	yes		2
table: episode_events	ref	TEXT	yes		
table: episode_events	sort_key	TEXT	yes		
table: episode_events	receipt_ts	TEXT	yes		
table: episode_events	topic	TEXT	yes		
table: episode_events	kind	TEXT	yes		
table: episode_events	file	TEXT	yes		
table: episode_events	line_no	INTEGER	yes		
table: episode_events	role	TEXT	yes		
table: episode_events	late	INTEGER	yes	0	
table: episode_events	msg_id	TEXT	no		
table: episode_events	causation_id	TEXT	no		
table: episode_events	correlation_id	TEXT	no		
table: episode_events	topic_seq	INTEGER	no		
table: episode_events	eseq	INTEGER	no		
table: episode_events	publisher_run	INTEGER	yes	0	
table: episode_events	payload	TEXT	yes		
table: episodes	episode_id	INTEGER	no		1
table: episodes	kind	TEXT	yes		
table: episodes	anchor_key	TEXT	yes		
table: episodes	correlation_id	TEXT	no		
table: episodes	root_msg_id	TEXT	no		
table: episodes	run_seq	INTEGER	no		
table: episodes	symbol	TEXT	no		
table: episodes	actor	TEXT	no		
table: episodes	opened_sort_key	TEXT	yes		
table: episodes	closed_sort_key	TEXT	no		
table: episodes	opened_ts	TEXT	yes		

Object	Field	Declared type	Not null	Default	Primary key
table: episodes	closed_ts	TEXT	no		
table: episodes	outcome	TEXT	no		
table: episodes	summary	TEXT	yes		
table: episodes	facts_json	TEXT	yes		
table: links	from_episode	INTEGER	yes		1
table: links	to_episode	INTEGER	yes		2
table: links	relation	TEXT	yes		3
table: links	confidence	TEXT	yes		
table: links	evidence	TEXT	yes		
table: replay_meta	key	TEXT	no		1
table: replay_meta	value	TEXT	yes		
table: stated_links	from_ref	TEXT	yes		1
table: stated_links	to_ref	TEXT	yes		2
table: stated_links	relation	TEXT	yes		3
table: stated_links	confidence	TEXT	yes		
table: stated_links	evidence	TEXT	yes		

5.10.2 DDL

table actors

```
CREATE TABLE actors (
    gateway_id   TEXT PRIMARY KEY,
    description  TEXT,
    first_seen   TEXT NOT NULL,
    last_seen    TEXT NOT NULL,
    orders       INTEGER NOT NULL DEFAULT 0,
    trades       INTEGER NOT NULL DEFAULT 0
)
```

table anomalies

```
CREATE TABLE anomalies (
    anomaly_id  INTEGER PRIMARY KEY,
    code        TEXT NOT NULL,
    severity    TEXT NOT NULL,
    episode_id  INTEGER REFERENCES episodes(episode_id),
    sort_key    TEXT NOT NULL,
    receipt_ts  TEXT NOT NULL,
    detail      TEXT NOT NULL,
    file        TEXT,
    line_no     INTEGER NOT NULL DEFAULT 0
)
```

table episode_events

```
CREATE TABLE episode_events (
    episode_id  INTEGER NOT NULL REFERENCES episodes(episode_id),
    seq_in_ep   INTEGER NOT NULL,
    ref         TEXT    NOT NULL,
    sort_key    TEXT    NOT NULL,
    receipt_ts  TEXT    NOT NULL,
    topic       TEXT    NOT NULL,
    kind        TEXT    NOT NULL,
    file        TEXT    NOT NULL,
    line_no     INTEGER NOT NULL,
    role        TEXT    NOT NULL,
    late        INTEGER NOT NULL DEFAULT 0,
    msg_id         TEXT,
    causation_id   TEXT,
    correlation_id TEXT,
    topic_seq      INTEGER,
    eseq           INTEGER,
    publisher_run  INTEGER NOT NULL DEFAULT 0,
    payload     TEXT    NOT NULL,
    PRIMARY KEY (episode_id, seq_in_ep)
)
```

table episodes

```
CREATE TABLE episodes (
    episode_id      INTEGER PRIMARY KEY,
    kind            TEXT NOT NULL,
    anchor_key      TEXT NOT NULL,
    correlation_id  TEXT,
    root_msg_id     TEXT,
    run_seq         INTEGER,
    symbol          TEXT,
    actor           TEXT,
    opened_sort_key TEXT NOT NULL,
    closed_sort_key TEXT,
    opened_ts       TEXT NOT NULL,
    closed_ts       TEXT,
    outcome         TEXT,
    summary         TEXT NOT NULL,
    facts_json      TEXT NOT NULL
)
```

table links

```
CREATE TABLE links (
    from_episode  INTEGER NOT NULL REFERENCES episodes(episode_id),
    to_episode    INTEGER NOT NULL REFERENCES episodes(episode_id),
    relation      TEXT NOT NULL,
    confidence    TEXT NOT NULL,
    evidence      TEXT NOT NULL,
    PRIMARY KEY (from_episode, to_episode, relation)
)
```

table replay_meta

```
CREATE TABLE replay_meta (
    key         TEXT PRIMARY KEY,
    value       TEXT NOT NULL
)
```

table stated_links

```
CREATE TABLE stated_links (
    from_ref    TEXT NOT NULL,
    to_ref      TEXT NOT NULL,
    relation    TEXT NOT NULL,
    confidence  TEXT NOT NULL,
    evidence    TEXT NOT NULL,
    PRIMARY KEY (from_ref, to_ref, relation)
)
```

index idx_anom_code

```
CREATE INDEX idx_anom_code     ON anomalies(code, sort_key)
```

index idx_ee_cause

```
CREATE INDEX idx_ee_cause      ON episode_events(causation_id)
```

index idx_ee_chain

```
CREATE INDEX idx_ee_chain      ON episode_events(correlation_id)
```

index idx_ee_msg

```
CREATE INDEX idx_ee_msg        ON episode_events(msg_id)
```

index idx_ee_ref

```
CREATE INDEX idx_ee_ref        ON episode_events(ref)
```

index idx_ee_sort

```
CREATE INDEX idx_ee_sort       ON episode_events(sort_key)
```

index idx_ep_actor_ts

```
CREATE INDEX idx_ep_actor_ts   ON episodes(actor, opened_ts)
```

index idx_ep_chain

```
CREATE INDEX idx_ep_chain      ON episodes(correlation_id)
```

index idx_ep_kind_key

```
CREATE INDEX idx_ep_kind_key   ON episodes(kind, anchor_key)
```

index idx_ep_sort

```
CREATE INDEX idx_ep_sort       ON episodes(opened_sort_key)
```

index idx_ep_symbol_ts

```
CREATE INDEX idx_ep_symbol_ts  ON episodes(symbol, opened_ts)
```

index idx_links_to

```
CREATE INDEX idx_links_to      ON links(to_episode)
```

CHAPTER 6

The Order Book: Deep Dive

Learning objectives

After working through this chapter you will be able to:

- **Place `OrderBook` in the engine** — know exactly which engine function hands it an order, what it returns, and why it never touches the network
- **Read the data structures** — the four heaps, the trailing-stop list and the **five side indexes**, and explain which question each one answers
- **Explain price-time priority in code** — how the tuple key `(±price, arrival_seq)` turns Python's min-heap into a fair queue, and why `arrival_seq` (not a timestamp) decides who is first
- **Explain lazy deletion** — why a cancel does not remove anything from a heap, who throws the garbage away (`_peek`, `_compact_heaps`), and why `len(heap)` is not the number of orders
- **Trace one incoming order** through `process` → `_match_*` → `_sweep` → `_apply_fill` → `_rest` → stop cascade, and predict every trade and event it produces
- **Handle the corner cases** — icebergs (slice cap and refill to the back), self-match prevention, stops and trailing stops, FOK pre-checks, OCO/quote callbacks, and the no-matching (auction) mode
- **Keep the book consistent when you change it** — know the synchronisation rules, the invariants the test-suite checks, and which functions you must touch together
- **Learn by debugging** — drive a book from a ten-line script or from a test, set the right breakpoints and watch one order flow through

How to read this chapter

Open `src/edumatcher/engine/order_book.py` side by side with this page. The module docstring at the top of that file is a compressed version of this chapter, and every section below quotes the real code. Function names are used instead of line numbers because line numbers change: use your editor's *Go to Symbol* (`Ctrl+Shift+O` / `Cmd+Shift+O` in VS Code) to jump to, for example, `_sweep`.

All prices inside the book are **integer ticks** (for a symbol with two decimals, `10010` means `100.10`). Conversion to display money happens only at the edge (`from_ticks`), so dictionary keys and arithmetic are exact.

6.1 Where the File Sits

The engine process (`engine/main.py`) owns the network side; `OrderBook` owns the *rules of matching* and knows nothing about ZeroMQ. The module docstring draws the pipeline like this:

```
gateway --PUSH--> Engine.run()                  (receive + decode)
                  Engine._dispatch_pull_message  (route by topic)
                  Engine._handle_new_order       (validate, ACK, gate on session)
                  OrderBook.process(order)       <== YOU ARE HERE
                      |  returns (trades, events)
                  Engine._publish_dispatch       (order.fill / order.cancelled /
                                                  reject, then trade.executed)
                  --PUB--> gateways, market data, clearing, audit ...
```



Three properties follow from this split, and they shape everything in the file:

1. **One book per symbol.** The engine keeps one `OrderBook` instance per symbol (`Engine._book(symbol)` creates it on first use). Books never interact with each other.
2. **Synchronous and single-threaded.** `process()` runs to completion before the engine looks at the next message. There are no locks because there is no concurrency inside the book.
3. **It never publishes.** The book *mutates* `Order` objects in place and *returns* what happened as two lists: `trades` (new `Trade` objects) and `events` (orders whose state changed). The engine turns those into bus messages. This is what makes the matching rules unit-testable without a network — most tests in `tests/` build an `OrderBook`, call `process()` and assert on the two lists.

By the time an order reaches `process()` the engine has already authenticated the gateway, checked the symbol, validated quantity and price, applied the price collar and the session/halt gates, and (usually) sent the positive ACK. The book only decides *how the order trades*. Those engine-side checks are covered in Hot-Path Risk Checks near the end of this chapter.

6.2 What Is an Order Book?

An exchange is a marketplace where buyers and sellers meet. The order book is the exchange's record of every unmet offer — all the buyers who want to buy but have not found a seller yet, and all the sellers who want to sell but have not found a buyer.

```
                THE ORDER BOOK FOR AAPL

┌──────────────────────────────────────────────────┐
│  BUY SIDE (BIDS)           SELL SIDE (ASKS)      │
│                                                  │
│  $150.28 × 200 shares      $150.31 × 500 shares  │
│  $150.27 × 100 shares      $150.33 × 200 shares  │
│  $150.25 × 400 shares      $150.35 × 300 shares  │
│          ↑                        ↑              │
│      Best bid                 Best ask           │
│      (highest buyer)          (lowest seller)    │
│                                                  │
│           SPREAD = $150.31 - $150.28 = $0.03     │
└──────────────────────────────────────────────────┘
```

When a new buy order arrives at $150.31 or higher, it *crosses the spread* and trades with the best ask. When a new buy order arrives at $150.27 it cannot trade (the cheapest seller wants $150.31), so it *rests* on the bid side and waits.

Two words are used throughout the code:

Term	Also called	Meaning
Aggressor	taker	The *incoming* order — the one being processed right now.
Passive	maker, resting order	An order already sitting on the book.

Trades happen at the passive order's price

A trade always prints at the **resting** order's price, never the aggressor's. A buy limit at 101 that meets a resting ask at 100 trades at 100 — the aggressor receives *price improvement*. In the code this is the single line `_apply_fill(aggressor, best, fill_qty, best_price, ...)` at the bottom of `_sweep`: `best_price` is the passive order's price.

The rest of this chapter is about how `OrderBook` stores the two sides so that "who is the best counterparty right now?" is cheap to answer, and how it keeps that storage consistent while orders arrive, trade, get amended and get cancelled.

6.3 Background: The Heap Data Structure

Matching only ever needs one question answered quickly: *which resting order must be matched next?* A **binary heap** answers exactly that question.

A heap is a binary tree stored in a plain list, with one rule: every parent is less than or equal to its children. Nothing else is ordered. The consequence is that the smallest item is always at index `0`, while the rest of the list is only *partially* sorted.

```
list:  [ 10, 20, 15, 40, 30, 50 ]          tree view:          10
index:    0   1   2   3   4   5                                /    \
                                                             20      15
children of i are at 2i+1 and 2i+2                          /  \    /
                                                           40  30  50
```

Python's `heapq` module implements the operations on an ordinary list:

Operation	`heapq` call	Cost	Used by `OrderBook` for
Look at the best item	`heap[0]`	O(1)	`_peek`, `_check_stops`
Insert	`heapq.heappush(heap, x)`	O(log n)	`_rest`, `_add_stop`, `_reinsert_iceberg`, `amend_order`
Remove the best item	`heapq.heappop(heap)`	O(log n)	`_peek` (dead entries), `_check_stops`
Build from a list	`heapq.heapify(heap)`	O(n)	`_compact_heaps`

Two limitations matter for the design:

1. **`heapq` is a *min*-heap.** The best ask (lowest price) comes out first naturally, but the best bid is the *highest* price. The trick is to store the price **negated**: the smallest `-price` is the largest price.
2. **A heap cannot cheaply remove an item from the middle.** Finding it is O(n) and repairing the tree is O(log n). Cancels hit the middle of the book all the time. The answer is *lazy deletion*, explained below.

Why a heap and not one sorted list per price level?

A common textbook design is a sorted map of price levels, each holding a FIFO queue. That design makes "remove from the middle" cheap but needs a sorted container (Python has none built in) and more bookkeeping per level. EduMatcher chooses the heap because pushing and popping the best order are both O(log n) with the standard library alone, and matching only ever needs "the best one". The price of that choice is lazy deletion and the side indexes described below.

6.4 The Priority Key: `(±price, arrival_seq)`

Every entry in a bid or ask heap carries a two-element tuple key. Python compares tuples element by element, so the first element decides and the second only breaks ties:

Heap	Key	`heap[0]` is
`_asks`	`( price_ticks, arrival_seq)`	the lowest ask; among equal prices the earliest
`_bids`	`(-price_ticks, arrival_seq)`	the highest bid; among equal prices the earliest

Together this is exactly **price-time priority**: better price first, then first come first served. The keys are built in `_rest`:

```
# OrderBook._rest
order.arrival_seq = self._next_seq()
if order.side == Side.BUY:
    key = (-price, order.arrival_seq)
    heap = self._bids
    self._bid_qty[price] = self._bid_qty.get(price, 0) + qty
else:
    key = (price, order.arrival_seq)
    heap = self._asks
    self._ask_qty[price] = self._ask_qty.get(price, 0) + qty
```

Real keys from a small book (three asks and one bid, produced by the script in Learning by Debugging):

```
_asks: [((10010, 1), a1), ((10010, 2), a2), ((10020, 3), a3)]
_bids: [((-10000, 4), b1)]
```

`a1` and `a2` share the price 100.10; `a1` wins the tie because it has the smaller `arrival_seq`. The bid key is negative because bids are stored negated.

6.4.1 `arrival_seq` is the clock, not `ts_ns`

`arrival_seq` comes from a per-book monotonic counter:

```
def _next_seq(self) -> int:
    """Return the next monotonic arrival sequence number."""
    self._arrival_seq += 1
    return self._arrival_seq
```

arrival_seq vs. the order's ts_ns — do not confuse them

An `Order` also carries `ts_ns`, a nanosecond timestamp that arrives with the order (see the ALF protocol and the message reference). It is tempting to think "time priority" means that timestamp. It does not.

A participant's clock can be skewed, and a value can be crafted, so the book **never** uses a client-supplied time to decide queue position — a back-dated timestamp cannot jump the queue. (The source comments call this the client "`timestamp`"; the field on `Order` is `ts_ns`.)

Queue position is decided by **`Order.arrival_seq`**, which the book itself assigns at the moment an order is placed on a heap. Because it is engine-assigned and strictly increasing, two orders can never share one and their order cannot be forged. This is **finding H1** in the source; see Order Types — Priority Rules for the participant-facing statement of the same rule.

`_next_seq()` is called in exactly four places — and each one is a moment where an order goes to the **back** of its queue:

Caller	When
`_rest`	an order starts resting on the bid/ask heap
`_add_stop`	a STOP / STOP_LIMIT is armed (ties between equal trigger prices)
`_reinsert_iceberg`	an iceberg shows a fresh slice (it loses time priority)
`amend_order`	an amend that raises the quantity or changes the price

`ts_ns`, on the other hand, is overwritten with the engine's batch clock when an order is re-queued (amend with priority loss, iceberg refill) or converted (a stop that triggers), so the published order reflects when that happened. It never takes part in a heap comparison.

6.5 `_HeapEntry`: Why the Heap Does Not Hold Orders

The heaps hold small wrapper objects, not `Orders`:

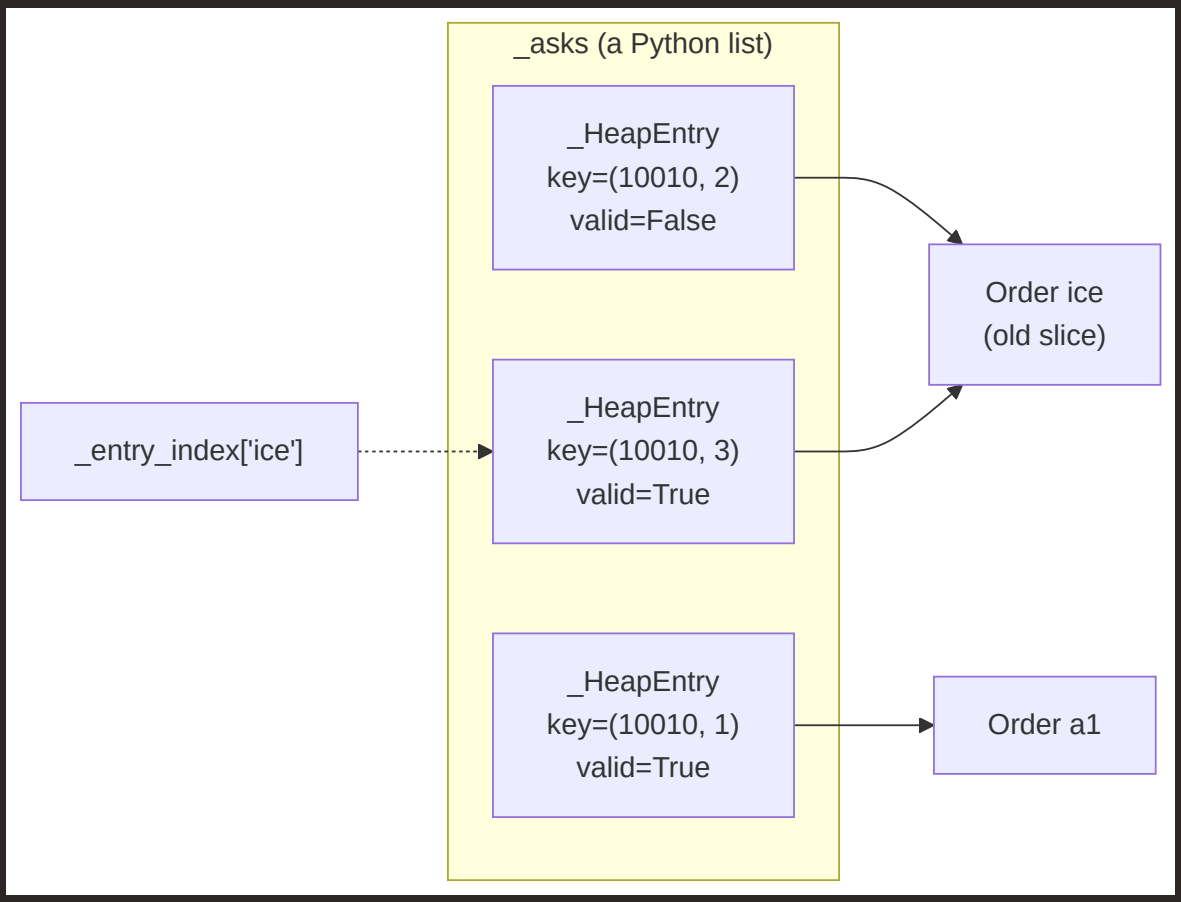
```
class _HeapEntry:
    __slots__ = ("key", "order", "valid")

    def __init__(self, key: tuple[Any, ...], order: Order, valid: bool = True) -> None:
        self.key = key
        self.order = order
        self.valid = valid

    def __lt__(self, other: "_HeapEntry") -> bool:
        return self.key < other.key
```

The class docstring gives three reasons, and each one is load-bearing:

1. **Defined ordering.** `heapq` orders items with `<`. Defining `__lt__` as "compare the keys" guarantees two `Order` objects are never compared directly (a dataclass `Order` has no sensible ordering, and comparing two would raise `TypeError`).
2. **The key is frozen at push time.** A heap is only correct if an item's key never changes while it is inside. An order's price or `arrival_seq` *can* change (amend, iceberg refill), so a changed order is pushed again as a **new** entry and the old entry is retired. The `Order` object itself is shared by both entries; only the newest entry is live.
3. **`valid` is the lazy-deletion flag.** Setting `entry.valid = False` is how the book "removes" an order from the middle of a heap in O(1). The dead entry is thrown away later (see Lazy Deletion).



The figure shows reason 2 in action: an iceberg (`ice`) that was refilled has two entries in the heap. The old one has `valid=False`; `_entry_index` points at the new one.

`__slots__` on hot objects

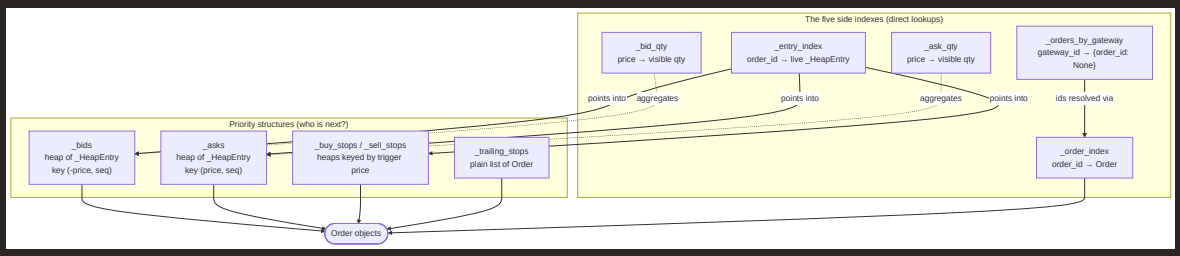
Both `_HeapEntry` and `OrderBook` declare `__slots__`. By default every Python instance carries a per-instance `__dict__`; `__slots__` replaces it with fixed attribute slots, which makes the object smaller and attribute access faster, and turns a typo such as `entry.vaild = False` into an `AttributeError` instead of a silently created attribute. A busy book creates one `_HeapEntry` for every resting order, amend and iceberg refill, so the saving adds up. See `docs-design/perf-notes.md` for the measurements behind this and the other hot-path micro-optimisations (cached locals in `_sweep`, the shared `now` timestamp, the module-level `_DEAD_STATUSES` frozenset).

6.6 The `OrderBook` Object at a Glance

`OrderBook.__slots__` lists every attribute the book owns. Grouped by purpose:

Group	Attributes
Identity	`symbol`
The book itself (heaps)	`_bids`, `_asks`
Dormant stops	`_buy_stops`, `_sell_stops` (heaps), `_trailing_stops` (list), `_has_stops` (fast-path flag), `_triggered` (waiting room)
The five side indexes	`_order_index`, `_entry_index`, `_orders_by_gateway`, `_bid_qty`, `_ask_qty`
Market statistics	`last_trade_price`, `last_trade_id`, `last_trade_qty`, `last_buy_price`, `last_sell_price`, `recent_trades`, `daily_qty`, `daily_value_ticks`, `daily_trades`
Counters	`_gtc_count` (resting orders with TIF=GTC, kept in step with `_orders_by_gateway`)
Engine hooks	`on_quote_leg_fill`, `on_oco_leg_done` (callbacks), `trace` (shared report list)
Time priority	`_arrival_seq`

The figure below shows how the heaps and the five side indexes all point at the **same** `Order` objects. Nothing is copied: an `Order` lives once in memory and every structure holds a reference to it.



6.7 Why Five Side Indexes?

A heap answers exactly one question fast: *who is best?* Every other question the engine asks of a book would need a full O(n) walk of the heap — and the heap also contains garbage (see Lazy Deletion), so the walk would need to filter as well. Each side index exists to answer one of those other questions in O(1) or O(levels) instead:

Index	Type	Question it answers	Who asks	Without it
`_order_index`	`dict[str, Order]`	"Is order *id* resting here, and where is the object?"	`cancel_order`, `amend_order`, `get_order`, `resting_orders` (persistence, TIF expiry), `orders_for_gateway`	Scan two book heaps, two stop heaps and the trailing list on every cancel

Index	Type	Question it answers	Who asks	Without it
`_entry_index`	`dict[str, _HeapEntry]`	"Which heap entry is this order's *live* one?"	`cancel_order`, `amend_order`, `_smp_cancel_resting`, `_reinsert_iceberg`, `_purge_from_indexes`; `len()` = live entries for `_compact_heaps` / `book_stats`	No O(1) way to flip `valid`; and after an amend or iceberg refill there are *several* entries for one order — only this index knows which one is current
`_orders_by_gateway`	`dict[str, dict[str, None]]`	"What does gateway *X* have resting in this book?"	gateway disconnect, kill switch, cancel-by-symbol, quote reissue, `resting_count_by_gateway`	Scan every resting order of every gateway in every book during an emergency mass-cancel
`_bid_qty`	`dict[int, int]`	"How much is *shown* at bid price *p*? What is the best bid?"	`best_bid_ticks`, `depth_snapshot`, `book_stats.bid_levels`	Walk every bid entry to aggregate levels
`_ask_qty`	`dict[int, int]`	Same for the ask side	`best_ask_ticks`, `depth_snapshot`, `book_stats.ask_levels`	Walk every ask entry

The cost of having five extra structures is the **synchronisation rule**: every code path that changes the book must update all of them consistently. The module docstring calls this "the most common source of bugs when this file is edited". The Synchronisation Rules section lists exactly who updates what.

6.7.1 `_order_index` — the id lookup

```
# order_id → Order for every resting order (stops included): lets a
# cancel / amend find its order without scanning a heap.
self._order_index: dict[str, Order] = {}
```

It holds **every resting order**: plain and quote-leg LIMIT/ICEBERG orders, armed STOP/STOP_LIMIT orders and trailing stops. It does *not* hold the incoming aggressor while it is being matched (it is not resting yet), nor terminal orders: a FILLED or CANCELLED order is purged immediately (finding **H7**), so `len(self._order_index)` is the number of live resting orders and `resting_orders()` stays O(resting) rather than O(everything ever seen).

6.7.2 `_entry_index` — the handle for lazy deletion

```
# order_id → the order's CURRENT live _HeapEntry (plain/quote orders
# and stop/stop-limit orders; trailing stops have none).  Cancelling
# flips entry.valid to False: this is the handle for lazy deletion.
self._entry_index: dict[str, _HeapEntry] = {}
```

Because a heap cannot find an item, the book keeps the entry object itself. Cancel is then `self._entry_index[order_id].valid = False` — O(1). Whenever an order is re-pushed (amend with priority loss, iceberg refill) this index is overwritten with the *new* entry, which is how the book tells the live entry apart from retired ones. Trailing stops are absent because they live in a plain list, not a heap.

6.7.3 `_orders_by_gateway` — "everything gateway X has here"

```
self._orders_by_gateway: dict[str, dict[str, None]] = {}
```

Why dict[str, None] and not set[str]?

It is used as an **insertion-ordered set**. A `set` of strings iterates in hash order, and Python randomises string hashes per process. A mass-cancel that iterated a set would therefore publish its cancels in a different order on every run, making the exchange non-deterministic. A dict iterates in insertion order, so two runs over the same input publish identically (finding

X11). The code adds with `setdefault(gw, {})[order.id] = None` and removes with `.pop(order.id, None)`.
Adds and removes go through two helpers, `_track_gateway(order)` and `_discard_from_gateway_index(order)`, so the bookkeeping exists once. They also maintain `_gtc_count`, the number of resting orders with `TIF.GTC` (the orders that survive the close), which lets `book_stats()` report it without walking the book. Both are idempotent: an id already present is not counted twice, and a missing one is ignored.
`orders_for_gateway(gw)` resolves the ids through `_order_index`, and `quote_orders_for_gateway(gw)` filters that result to `OrderOrigin.QUOTE`. The engine calls them from `_handle_gateway_disconnect`, `_handle_kill_switch` and the quote-reissue path. When a gateway's last order leaves the book, `_discard_from_gateway_index` deletes the gateway key too, so `resting_count_by_gateway()` reports only gateways that actually have orders. For why this matters on a real venue, see the guided tour, §10.

6.7.4 `_bid_qty` / `_ask_qty` — display depth per price level

```
# Price-level quantity index: price → total visible resting qty
# (an iceberg contributes only its displayed slice).  Display depth,
# not the matching source of truth: matching walks the heaps.
self._bid_qty: dict[int, int] = {}
self._ask_qty: dict[int, int] = {}
```

Three rules define these dictionaries:

1. **Visible quantity only.** A plain order contributes `remaining_qty`; an iceberg contributes only `displayed_qty` (its currently shown slice), never its hidden remainder. This is what market data is allowed to show.
2. **A level at zero is deleted.** `_deduct_qty_index` pops the key when the quantity reaches zero, so "key present" means "level exists". That is what `best_bid_ticks()` (`max` over the keys) and `bid_levels` rely on.
3. **Not used for matching.** `_sweep` walks the heaps. Because of rule 1 the index *under-counts* liquidity when icebergs rest, which is why the FOK pre-check (`_fok_available_qty`) and the combo pre-check (`fillable_quantity`) walk the heap entries instead.

Why two dicts instead of one keyed by `(side, price)`? The two sides are queried separately (best bid vs best ask, bid depth vs ask depth), and during an auction the book can legitimately be *crossed*, with a bid and an ask at the same price.

The quantity updates all go through a handful of places: `_rest` and `_reinsert_iceberg` add, `_deduct_qty_index` subtracts, and `amend_order` does both:

```
def _deduct_qty_index(self, order: Order, qty: int) -> None:
    if order.price_ticks is None or qty <= 0:
        return
    if order.side == Side.BUY:
        idx = self._bid_qty
    else:
        idx = self._ask_qty
    current = idx.get(order.price_ticks, 0)
    updated = current - qty
    if updated <= 0:
        idx.pop(order.price_ticks, None)
    else:
        idx[order.price_ticks] = updated
```

6.7.5 Which order lives where

Order state	Heap / list	`_order_index`	`_entry_index`	`_orders_by_gateway`	`_bid_qty`/`_ask_qty`
Incoming aggressor being matched	—	—	—	—	—
Resting LIMIT / quote leg	`_bids` / `_asks`				`remaining_qty`

Order state	Heap / list	_order_index	_entry_index	_orders_by_gateway	_bid_qty/_ask_qty
Resting ICEBERG	_bids / _asks				displayed_qty
Armed STOP / STOP_LIMIT	_buy_stops / _sell_stops				—
Armed TRAILING_STOP	_trailing_stops		—		—
Triggered stop waiting its turn	— (in _triggered)	—	—	—	—
FILLED / CANCELLED / REJECTED	maybe a dead entry	—	—	—	—

A STOP_LIMIT carries a limit price, but while it is dormant it must **not** contribute to the level index; `cancel_order` has an explicit guard for that, otherwise cancelling a stop would subtract quantity from genuine orders that happen to rest at the same price.

6.8 The Rest of the State

6.8.1 Stop structures

```
self._buy_stops: list[_HeapEntry] = []   # key ( stop_price, arrival_seq)
self._sell_stops: list[_HeapEntry] = []  # key (-stop_price, arrival_seq)
self._trailing_stops: list[Order] = []
```

A BUY stop fires when the price **rises** to its trigger, so the one closest to firing is the *lowest* trigger: a min-heap. A SELL stop fires when the price **falls**, so the closest is the *highest* trigger: a max-heap via negation. In both cases `heap[0]` is the next stop to fire, so a trade that triggers nothing costs a single comparison. Trailing stops are a plain list because every trade can move every trailing trigger, which would break a heap's frozen keys.

Two more fields support the stop machinery:

- `_has_stops` — `True` while any stop structure is non-empty. `process()` tests it before calling the stop checks, so a book without stops pays nothing per trade. `_check_stops` / `_check_trailing_stops` reset it when all three structures are empty.
- `_triggered: dict[str, Order]` — the **waiting room** for stops that have fired but not yet been matched (see Stop Orders). They are out of every index but must still be cancellable, so `cancel_order` looks here as well as in `_order_index`.

6.8.2 Market statistics

```
self.last_trade_price: Optional[int] = None
self.last_trade_qty: Optional[int] = None
self.last_buy_price: Optional[int] = None   # last trade where buyer was aggressor
self.last_sell_price: Optional[int] = None  # last trade where seller was aggressor
self.recent_trades: deque[Trade] = deque(maxlen=recent_trades_maxlen)  # default 20
self.daily_qty: int = 0
self.daily_value_ticks: int = 0
self.daily_trades: int = 0
```

All of them are written in exactly one place, `_apply_fill`. The same function also records `last_trade_id`, the id of the trade that set `last_trade_price`: when a stop fires, that is the trade it triggered on. Their uses are covered in The Three Last Prices. `daily_value_ticks` accumulates `price_ticks × qty` as an integer (finding **M11**); the `daily_value` property converts it to money once, so no floating-point drift accumulates over thousands of fills.

6.8.3 Engine hooks

```
self.on_quote_leg_fill: Optional[Callable[[Order], None]] = None
self.on_oco_leg_done: Optional[Callable[[Order], None]] = None
self.trace: Optional[list[tuple[Any, ...]]] = None
```

The book cannot cancel a market maker's other quote leg or an OCO sibling by itself — that bookkeeping lives in the engine. So `Engine._book()` wires two callbacks, and a trace list, when it creates a book:

```
# Engine._book (engine/main.py)
book = OrderBook(symbol, recent_trades_maxlen=self.recent_trades_maxlen)
book.on_quote_leg_fill = self._on_quote_leg_filled   # X3
book.on_oco_leg_done = self._check_oco_after_event   # X15
book.trace = self._trace                             # S24: one list, shared by all books
```

The callbacks are called *during* the sweep, which is explained in `_apply_fill`. In a plain unit test they are `None` and simply skipped. `trace` is covered in Trace Records.

6.9 Lazy Deletion and Compaction

Cancelling an order deep inside a heap would cost O(n). Instead the book **does not remove it**: it sets `entry.valid = False` and leaves the dead entry where it is. Whoever next looks at the top of the heap discards dead entries until it finds a live one. That is `_peek`:

```
def _peek(self, heap: list[_HeapEntry]) -> Optional[Order]:
    while heap:
        entry = heap[0]
        if not entry.valid:
            heapq.heappop(heap)
            continue
        o = entry.order
        if o.status in _DEAD_STATUSES:
            heapq.heappop(heap)
            entry.valid = False
            continue
        return o
    return None
```

There are two kinds of dead entry:

- `valid == False` — the order was cancelled, filled, amended or refilled. In the last two cases the *order* is still alive under a newer entry.
- a valid entry whose order is in a terminal status (`_DEAD_STATUSES` = FILLED, CANCELLED, REJECTED, EXPIRED). This is the safety net.

`_peek` returns the live order **without popping it**: the caller decides what happens to it, and a fully filled order is retired by `_apply_fill`. Each dead entry is popped at most once, so `_peek` is amortised O(log n).

6.9.1 Garbage that never reaches the top

An order cancelled at a price far from the market may never reach `heap[0]`. Without help, those entries — each keeping its `Order` alive — would accumulate for the life of the process (an AI swarm cancels thousands of stale orders a minute). `_compact_heaps` fixes that, and is called from `_purge_from_indexes` every time an entry is retired, and after the two paths that retire an entry and push its replacement (an amend that loses priority and `_reinsert_iceberg`), so the heaps never exceed the bound for long:

```
_COMPACT_FACTOR = 2
_COMPACT_SLACK = 64

def _compact_heaps(self) -> None:
    heaps = (self._bids, self._asks, self._buy_stops, self._sell_stops)
    if sum(map(len, heaps)) <= (
        _COMPACT_FACTOR * len(self._entry_index) + _COMPACT_SLACK
    ):
        return
    for heap in heaps:
        heap[:] = [
            e for e in heap if e.valid and e.order.status not in _DEAD_STATUSES
        ]
        heapq.heapify(heap)
```

Once the four heaps together hold more than *twice the live entries plus 64*, all four are filtered and re-heapified in O(n). Because a rebuild only happens after at least as many retirements as there are live entries, the cost is amortised O(1) per removal. The slack of 64 keeps a nearly empty book from rebuilding after every single cancel. `heap[:] = ...` replaces the list *contents* in place, so every reference to the list stays valid.

len(self._bids) is NOT the number of resting bids

It includes dead entries. Use `len(book._entry_index)` (live entries in all four heaps), `len(book._order_index)` (all resting orders, including trailing stops) or `book.book_stats()`. `BookStats.heap_bloat_ratio` reports the heap size over the `2 × live + 64` that `_compact_heaps` tolerates (1.0 or less when healthy), and the health beacon publishes it so an operator can see whether compaction keeps up. The dead share itself is no signal: it legitimately climbs to about half the heap before each rebuild. `tests/test_order_book_compaction.py` pins this behaviour.

6.9.2 `_purge_from_indexes` — the single exit door

```
def _purge_from_indexes(self, order: Order) -> None:
    entry = self._entry_index.pop(order.id, None)
    if entry is not None:
        entry.valid = False
        self._compact_heaps()
    self._order_index.pop(order.id, None)
    self._discard_from_gateway_index(order)
```

Every way a resting order can *end* — `cancel_order`, a full fill in `_apply_fill` (aggressor or passive), and `_smp_cancel_resting` — funnels through this function, so the three id-keyed indexes can never disagree about a terminal order. It is idempotent and safe to call for a taker that never rested (the pops find nothing). Note that it does **not** touch `_bid_qty`/`_ask_qty`: the callers deduct the quantity first, because only they know how much was visible.

6.9.3 Seen in a real run

Starting from the small book shown earlier, `a3` (500 @ 100.20) is cancelled:

```
before:  _asks: [((10010, 2), a2, valid=True), ((10020, 3), a3, valid=True)]
         _ask_qty: {10010: 100, 10020: 500}
cancel_order(a3.id)
after:   _asks: [((10010, 2), a2, valid=True), ((10020, 3), a3, valid=False)]
         _ask_qty: {10010: 100}
         a3 is gone from _order_index, _entry_index and _orders_by_gateway
```

The heap still has two entries; only one is live. The level 100.20 has disappeared from `_ask_qty` because its quantity reached zero.

6.10 Synchronisation Rules

This table is the contract every change to `order_book.py` must respect. Each row is one code path; each column one structure. "—" means the path must leave that structure alone.

Code path	Heap / list	`_bid_qty`/`_ask_qty`	`_order_index`	`_entry_index`	`_orders_by_gateway`
`_rest` (LIMIT/ICEBERG starts resting)	push new entry	`+=` visible	add	add	add (`_track_gateway`)
`_add_stop` (STOP/STOP_LIMIT armed)	push on stop heap	—	add	add	add
`_add_trailing_stop`	append to list	—	add	—	add
`_apply_fill`, passive partly filled	—	`-=` fill	—	—	—
`_apply_fill`, passive or aggressor FILLED	entry invalidated	`-=` fill (passive)	purge	purge	purge
`_apply_fill`, iceberg slice used up	old entry invalid, new entry pushed (`_reinsert_iceberg`)	`-=` fill, `+=` new peak	—	replaced	—
`cancel_order`	entry invalidated	`-=` visible (not for stops)	purge	purge	purge
`_smp_cancel_resting`	entry invalidated	`-=` visible	purge	purge	purge
`amend_order`, priority kept	— (same entry)	`-=` old visible, `+=` new visible	—	—	—
`amend_order`, priority lost	old entry invalid, new entry pushed	`-=` old visible, `+=` new visible	—	replaced	—
`_check_stops` (stop fires)	popped from stop heap	—	pop	pop	discard
`_check_trailing_stops` (fires)	dropped from list	—	pop	—	discard
`_trigger_stop_now` (breached on entry)	— (never armed)	—	—	—	—
`_peek`	pops dead entries	—	—	—	—
`_compact_heaps`	rebuilt without dead entries	—	—	—	—

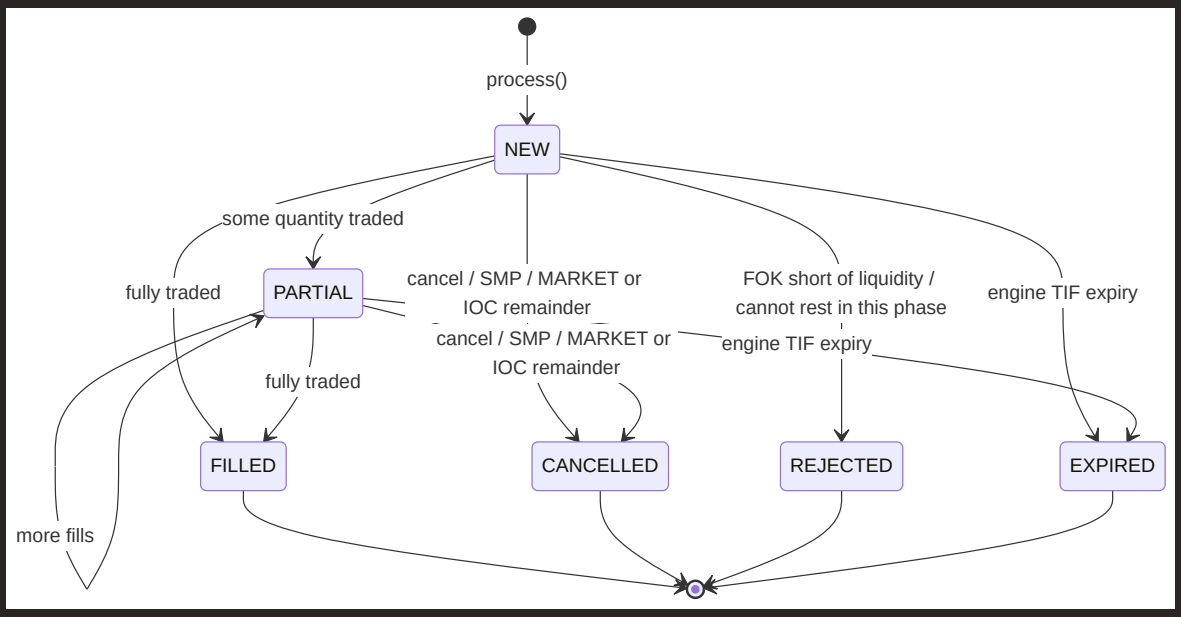
The invariants the tests enforce

`tests/engine_invariants.py` turns these rules into executable checks. `assert_book_invariants(book)` verifies:

- **I1** — `_bid_qty`/`_ask_qty` equal the sum of live visible quantity per level, recomputed from the heaps;
- **I2** — the book is not crossed in continuous trading;
- **I3** — per-order quantity sanity (`remaining_qty` within `[0, quantity]`, an iceberg's `displayed_qty` within its remainder);
- **I4** — every live heap entry is the very object `_entry_index` holds;
- **I5** — hygiene, checked when called with `include_hygiene=True`: no terminal order lingers in an index (H7).

`tests/test_book_invariants_random_ops.py` throws random sequences of orders, cancels and amends at a book and calls these checks after every step. If you change the book, run it.

6.11 The Order Lifecycle



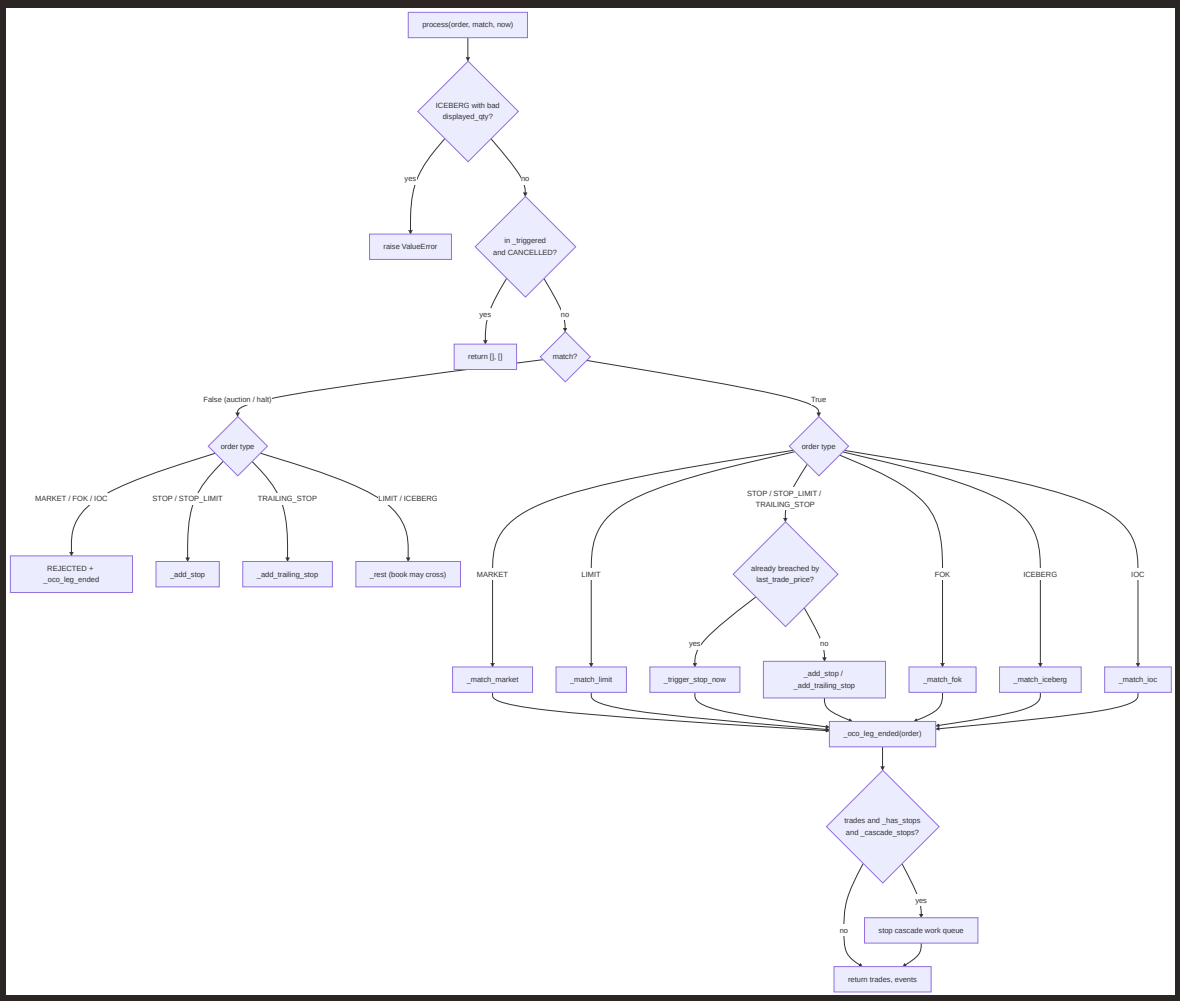
- **FILLED, CANCELLED, REJECTED and EXPIRED are terminal** (`_DEAD_STATUSES`). A terminal order is purged from every index (`_purge_from_indexes`) and its heap entry is invalidated.
- The book itself never sets EXPIRED. The engine's `_expire_tif` calls `book.cancel_order()` for every resting order of a given time-in-force (DAY orders at the close, ATO/ATC orders when their auction window ends) and then relabels the returned order as EXPIRED.
- A cancelled order **keeps its unfilled `remaining_qty`**. So `remaining_qty > 0` does not mean "should rest" — always test `status` (finding **C3**, see `_match_limit`).
- `cancel_reason` explains a cancel made by the book: `"SELF_MATCH_PREVENTED"` (SMP) or `"INSUFFICIENT_LIQUIDITY"` (MARKET/IOC remainder).

6.12 The Entry Point: `process()`

`process(order, *, match=True, now=None, _cascade_stops=True)` is the only way an order enters the book. The engine calls it from `_handle_new_order` as `book.process(order, match=do_match, now=now)`, where `do_match` is false while the session is in an auction/call phase or the symbol is halted.

The module docstring summarises the life of an incoming **LIMIT** order in seven steps. Keep this list next to you while reading the rest of the chapter:

1. **`process`** picks the matcher for the order type (`_match_limit`).
2. **`_match_limit`** calls `_sweep` against the *opposite* side's heap.
3. **`_sweep`** loops: look at the best opposite order (`_peek`), stop if its price is worse than our limit, apply self-match prevention, then take `min(our remaining, their available)`.
4. **`_apply_fill`**, for each such step, builds a `Trade`, updates both orders' `remaining_qty` / `status`, updates the price-level index, and appends *both* orders to `events`.
5. Back in **`_match_limit`**, whatever is left over **rests** on our own side (`_rest`). That is the difference between LIMIT and IOC/MARKET, which discard the remainder.
6. Back in **`process`**, every trade moved `last_trade_price`, so resting stops may now trigger (`_check_stops`); each triggered stop is itself processed as a new order.
7. **Return `(trades, events)`** to the engine.



6.12.1 Step by step through the code

1. Iceberg sanity guard. An iceberg whose `displayed_qty` is not in `(0, remaining_qty]` raises `ValueError`. A too-large slice would over-fill the order to a negative `remaining_qty`, after which every sweep pass fills zero and `while aggressor.remaining_qty > 0 and opposite_heap` never ends — the engine would hang. The ALF path already rejects such orders (engine check **M7**); the guard protects the paths that do not pass M7, such as restoring `gtc_orders.json` at startup or tests that build an `Order` by hand.

2. The triggered-stop waiting room.

```
if (
    self._triggered
    and self._triggered.pop(order.id, None) is not None
    and order.status == OrderStatus.CANCELLED
):
    return trades, events
```

A stop that fired earlier in this cascade waits in `_triggered` until it is processed. If its OCO sibling filled first and the engine cancelled it while it waited (finding **X15**), it must do nothing at all. For every ordinary new order the dict is empty and the whole test short-circuits on `self._triggered`.

3. One clock read. `now = now_ns()` is computed once (or passed in by the engine) and threaded to every `Trade.create()` and stop conversion, so a whole batch shares one timestamp.

4. No-matching mode (`match=False`). Used in call phases (opening/closing auction, reopening) and while a symbol is halted. Orders are *collected*, not traded:

```
if not match:
    if order.order_type in (OrderType.MARKET, OrderType.FOK, OrderType.IOC):
        order.status = OrderStatus.REJECTED
        events.append(order)
        self._oco_leg_ended(order)
        return trades, events
    if order.order_type in (OrderType.STOP, OrderType.STOP_LIMIT):
        self._add_stop(order, events)
        return trades, events
    if order.order_type == OrderType.TRAILING_STOP:
        self._add_trailing_stop(order, events)
        return trades, events
    self._rest(order)
    return trades, events
```

Anything that must trade *now* has nowhere to wait and is rejected. A TRAILING_STOP has no price and must go to the trailing list, never `_rest` (finding **C6**: without that branch a trailing stop submitted during a halt crashed the engine). LIMIT and ICEBERG rest without sweeping, so the book may become **crossed** (best bid best ask); the auction resolves that later (see No-Matching Mode and Auctions).

5. Continuous matching dispatches on `order.order_type` to one matcher (next section). Stops are special: if `last_trade_price` has *already* breached the trigger, the stop fires on entry via `_trigger_stop_now` instead of sleeping until some unrelated future trade (finding **M2**).

6. OCO notification. `self._oco_leg_ended(order)` tells the engine if the order just died (an IOC remainder, an SMP or FOK refusal) and belongs to an OCO group, so its sibling is pulled *before* the stop cascade can trade with it.

7. The stop cascade, driven by an explicit work queue in `run_stops`:

```
if trades and self._has_stops and _cascade_stops:
    sub_trades, sub_events = self.run_stops(now)
    trades.extend(sub_trades)
    events.extend(sub_events)
```

```
# OrderBook.run_stops
pending = self._check_stops(now) + self._check_trailing_stops(now)
while pending:
    t_order = pending.pop(0)
    sub_trades, sub_events = self.process(
        t_order, now=now, _cascade_stops=False
    )
    trades.extend(sub_trades)
    events.extend(sub_events)
    if sub_trades and self._has_stops:
        pending.extend(self._check_stops(now))
        pending.extend(self._check_trailing_stops(now))
```

The three guards in `process` are cheap early-outs: no trades means the price did not move; `_has_stops` means there is anything to trigger at all; `_cascade_stops` is `False` inside the nested call. That nested `self.process(...)` is **not** recursion of the cascade: the inner call cannot start its own loop, and whatever it triggers is appended to the same `pending` list, so the cascade runs breadth-first. An earlier version recursed, and a long enough stop chain exhausted Python's recursion limit (finding **#23**). `pending` is FIFO (`pop(0)`): stops run in the order they were released — buy stops first, then sell stops, then trailing stops, each group by trigger price and then arrival. Stops released by *later* trades queue **behind** those already waiting.

`run_stops` is public because `Engine._run_uncross` calls it too, after an auction print (see No-Matching Mode and Auctions). It returns `(trades, events)` in the same shape as `process`, and returns two empty lists at once when `_has_stops` is false.

6.12.2 What comes back

- `trades` — new `Trade` objects in execution order, including those produced by triggered stops.
- `events` — `Order` objects whose state changed. The **same** `Order` can appear several times (once per fill it took part in), and because the list holds references, every occurrence shows the order's *final* state. The engine de-duplicates in `_publish_dispatch` with a `seen` set.

The `_match_*` helpers return nothing: they `append` to the two lists they are given.

6.13 The Matchers

The `_match_*` methods differ only in (a) the price limit they hand to `_sweep` and (b) what they do with the part that did not trade. The source has this table above `_match_market`:

Type	Price limit	Unfilled remainder
MARKET	none	cancelled (`INSUFFICIENT_LIQUIDITY`)
LIMIT	order price	rests on the book
IOC	order price	cancelled (`INSUFFICIENT_LIQUIDITY`)
FOK	order price	never starts unless it can fill completely
ICEBERG	order price	rests, but only a small slice is displayed

In each, `opposite` is the heap the order trades **against**: a buyer lifts asks, a seller hits bids:

```
opposite = self._asks if order.side == Side.BUY else self._bids
```

6.13.1 LIMIT — `_match_limit`

```
def _match_limit(self, order, trades, events, now) -> None:
    opposite = self._asks if order.side == Side.BUY else self._bids
    self._sweep(order, opposite, price_limit=order.price_ticks,
                trades=trades, events=events, now=now)
    if order.remaining_qty > 0 and order.status not in _DEAD_STATUSES:
        self._rest(order)
```

The `status not in _DEAD_STATUSES` test is essential: an SMP cancel inside the sweep sets `status = CANCELLED` but leaves `remaining_qty > 0`. Resting it would put a dead order back in every index and leak phantom quantity into the level index (finding **C3**).

6.13.2 MARKET — `_match_market`

Sweeps with `price_limit=None` (any price). A remainder is never rested: the order becomes CANCELLED with `cancel_reason = "INSUFFICIENT_LIQUIDITY"` (unless SMP already set a more specific reason) and is appended to `events`. Without that append a totally unfilled MARKET order got a positive ACK and then silence (finding **#5**).

6.13.3 IOC — `_match_ioc`

Like LIMIT for the sweep, like MARKET for the remainder: partially filling is fine, the rest is cancelled with `INSUFFICIENT_LIQUIDITY` and reported.

6.13.4 FOK — `_match_fok`

Fill-or-kill is the only matcher that looks **before** it trades: a partial fill cannot be undone, so the decision is made up front by `_fok_available_qty`. It cannot use `_ask_qty`/`_bid_qty`, because those under-count icebergs, so it walks the real resting orders in priority order:

```
live = sorted(
    e
    for e in heap
    if e.valid
    and e.order.status not in _DEAD_STATUSES
    and e.order.price_ticks is not None
)
```

`sorted()` on a **copy** uses `_HeapEntry.__lt__`, giving true price-time order without disturbing the real heap (a heap list is only partially ordered, and popping would destroy the book). It is O(n log n), which is acceptable because FOK orders are rare. While walking, it applies the same rules the sweep will:

- **Hidden iceberg quantity counts**, but only *behind* everything else at that price level, because a refilled peak queues at the back (findings **#15**, **H8**).
- **Same-gateway orders do not count** when SMP is active; under `CANCEL_AGGRESSOR`/`CANCEL_BOTH` the walk *stops* at the first one, because the sweep would stop there too.
- **Only the first leg of an OCO pair counts** (X15): filling one leg pulls the other.

If the available quantity is short, the order never trades. SMP conflicts are resolved first (CANCEL_RESTING/CANCEL_BOTH cancel the conflicting resting orders), and then the FOK becomes either **CANCELLED** with `SELF_MATCH_PREVENTED` (it would have self-matched) or **REJECTED** (a genuine liquidity shortfall). A safety net after the sweep cancels any remainder, which should never fire given an accurate pre-check.

`fillable_quantity(side, price_limit)` is a simpler public cousin (counts all hidden quantity, no SMP); the engine's all-or-none combo pre-check uses it (finding **M5**).

6.13.5 ICEBERG — `_match_iceberg`

An **incoming** iceberg trades its *full* quantity like a LIMIT — hiding only matters for what rests. Whatever remains rests with a full peak:

```
if order.remaining_qty > 0 and order.status not in _DEAD_STATUSES:
    if order.visible_qty is not None:
        order.displayed_qty = min(order.visible_qty, order.remaining_qty)
    self._rest(order)
```

Iceberg vocabulary: `quantity` is the total size, `visible_qty` the fixed peak size chosen by the client, and `displayed_qty` how much of the current peak is still showing. See Icebergs for the resting behaviour.

6.13.6 STOP, STOP_LIMIT, TRAILING_STOP

These do not match on arrival unless already breached (M2). Normally they are armed in a stop structure; see Stop Orders.

6.14 The Matching Loop: `_sweep()`

This is the innermost loop of the exchange. It walks the opposite side from the best price to the worst (and, within a price, oldest first) and trades until one of four things stops it:

- the aggressor is completely filled (`remaining_qty == 0`);
- the opposite side is empty;
- the best remaining price is worse than `price_limit` (`None` = no limit);
- self-match prevention killed the aggressor.

```
_side = aggressor.side
_smp_action = aggressor.smp_action or SmpAction.NONE
_gw_id = aggressor.gateway_id
_peek = self._peek          # method lookup once
_apply_fill = self._apply_fill

while aggressor.remaining_qty > 0 and opposite_heap:
    # (1) best live counterparty (may be None: heap holds only dead entries)
    best = _peek(opposite_heap)
    if best is None:
        break
    best_price = best.price_ticks

    # (2) price check: heap is best-first, so first too-expensive => stop
    if price_limit is not None:
        if _side == Side.BUY and best_price > price_limit:
            break
        if _side == Side.SELL and best_price < price_limit:
            break

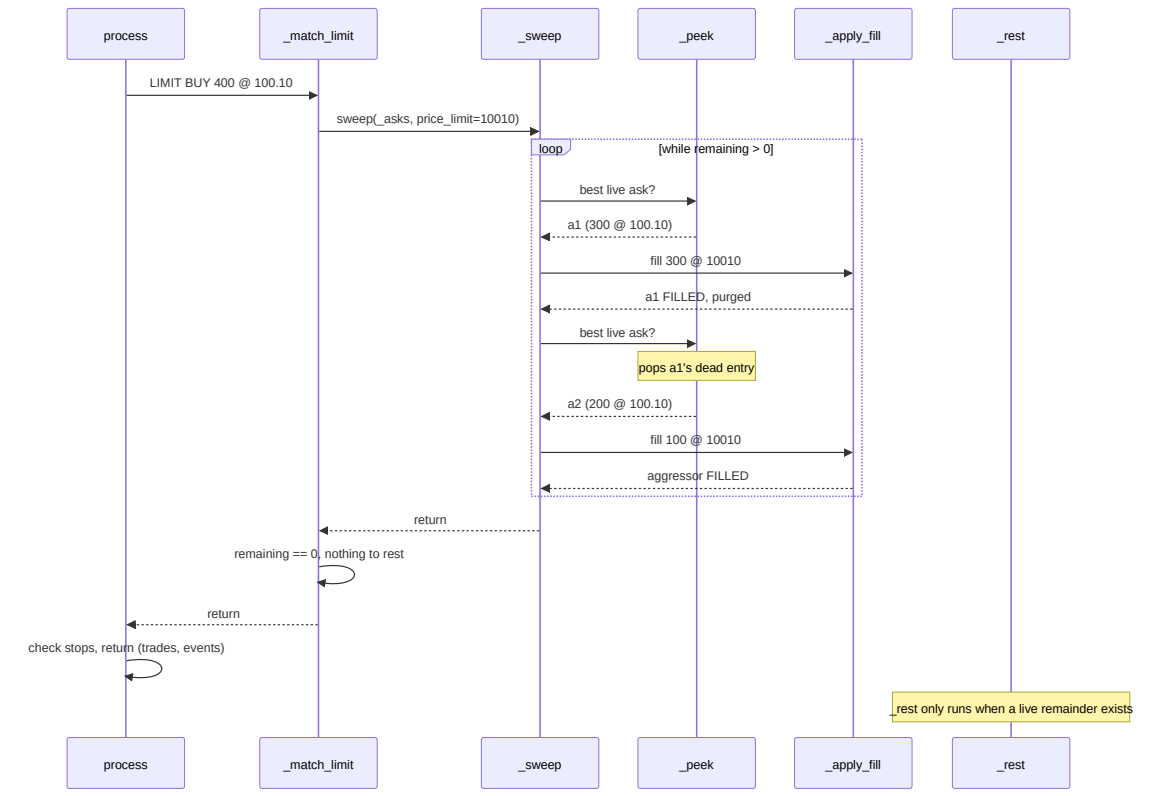
    # (3) self-match prevention (see the SMP section)
    if _smp_action != SmpAction.NONE and _gw_id == best.gateway_id:
        ...  # cancel aggressor / skip resting / both

    # (4) how much can trade with THIS order? passive iceberg: shown slice only
    _passive_avail = (
        best.displayed_qty
        if best.order_type == OrderType.ICEBERG and best.displayed_qty is not None
        else best.remaining_qty
    )
    fill_qty = min(aggressor.remaining_qty, _passive_avail)

    # (5) execute at the RESTING order's price
    _apply_fill(aggressor, best, fill_qty, best_price, trades, events, now)
```

Points worth noticing:

- **One trade per pass.** Each iteration trades the aggressor against exactly one resting order. A big aggressor produces a *sequence* of trades.
- **The loop never pops.** `_peek` returns the best live order; if `_apply_fill` fills it completely, `_purge_from_indexes` invalidates its entry and the next `_peek` throws it away and returns the next order.
- **Cached locals.** Attribute and method lookups are hoisted out of the loop (`_peek`, `_apply_fill`, `_side`) — a measurable gain in CPython; see `docs-design/perf-notes.md`.
- **The iceberg cap (finding #17).** Filling a passive iceberg's whole hidden `remaining_qty` in one go would jump the queue past same-price orders with better time priority. Capping at `displayed_qty` lets `_apply_fill` refill the slice at the back of the level, and the loop then naturally meets the other orders first.



6.15 Where Every Trade Is Born: `_apply_fill`

Every trade in the exchange — continuous or auction — is created here. Both `_sweep` and `auction.execute_uncross` call it. In order it:

1. creates the `Trade` (buyer and seller are worked out from the aggressor's side);
2. updates the book statistics (`last_trade_price`, which drives stops; daily volume; `recent_trades`);
3. reduces the aggressor's `remaining_qty`, sets PARTIAL/FILLED and appends it to `events`;
4. does the same for the passive order, including the iceberg refill and the level-index deduction;
5. fires the quote-leg / OCO callbacks.

```
if aggressor.side == Side.BUY:
    buy_order, sell_order = aggressor, passive
else:
    buy_order, sell_order = passive, aggressor

aggressor_side = _AUCTION_AGGRESSOR if both_resting else aggressor.side.value
trade = Trade.create(symbol=self.symbol, buy_order_id=buy_order.id,
                     sell_order_id=sell_order.id, ..., price=fill_price,
                     quantity=fill_qty, aggressor_side=aggressor_side, now=now,
                     trigger_order_id=(aggressor.id
                                       if not both_resting
                                       and aggressor.stop_price_ticks is not None
                                       else None))
trades.append(trade)
self.last_trade_price = fill_price
self.last_trade_id = trade.id
...
```

`trigger_order_id` marks a trade made by a **triggered stop**: the aggressor was a STOP / STOP_LIMIT / TRAILING_STOP that has since become a MARKET or LIMIT order but still carries its `stop_price_ticks`. Market data and the audit trail use it to show that this print was a stop cascade, not ordinary flow. Auction prints never carry it.

6.15.1 The aggressor side

```
aggressor.remaining_qty -= fill_qty
if aggressor.order_type == OrderType.ICEBERG:
    aggressor.displayed_qty = max(0, (aggressor.displayed_qty or 0) - fill_qty)
    if both_resting:
        ...  # auction only: deduct from the level index, maybe refill
elif both_resting:
    self._deduct_qty_index(aggressor, fill_qty)
aggressor.status = (
    OrderStatus.FILLED if aggressor.remaining_qty == 0 else OrderStatus.PARTIAL
)
if aggressor.status == OrderStatus.FILLED:
    self._purge_from_indexes(aggressor)
events.append(aggressor)
```

In continuous matching the aggressor **never rested**, so it has nothing in the level index to deduct. In an auction uncross (`both_resting=True`) both orders are resting, and leaving out the aggressor's deduction left phantom bid quantity behind (finding **#3**).

6.15.2 The passive side

```
passive.remaining_qty -= fill_qty
if passive.order_type == OrderType.ICEBERG:
    passive.displayed_qty = max(0, (passive.displayed_qty or 0) - fill_qty)
    if passive.remaining_qty > 0 and passive.displayed_qty == 0:
        new_peak = min(passive.visible_qty or 0, passive.remaining_qty)
        passive.displayed_qty = new_peak
        passive.ts_ns = now
        self._deduct_qty_index(passive, fill_qty)   # BEFORE the reinsert
        self._reinsert_iceberg(passive)
    else:
        self._deduct_qty_index(passive, fill_qty)
else:
    self._deduct_qty_index(passive, fill_qty)
passive.status = (
    OrderStatus.FILLED if passive.remaining_qty == 0 else OrderStatus.PARTIAL
)
if passive.status == OrderStatus.FILLED:
    self._purge_from_indexes(passive)
events.append(passive)
```

6.15.3 The callbacks

```
if not both_resting and self.on_quote_leg_fill is not None:
    if passive.quote_id is not None:
        self.on_quote_leg_fill(passive)
    if aggressor.quote_id is not None:
        self.on_quote_leg_fill(aggressor)
if not both_resting and self.on_oco_leg_done is not None:
    if passive.oco_group_id is not None and passive.status == OrderStatus.FILLED:
        self.on_oco_leg_done(passive)
    if aggressor.oco_group_id is not None and aggressor.status == OrderStatus.FILLED:
        self.on_oco_leg_done(aggressor)
```

A market maker's quote is a bid leg plus an ask leg; when one leg trades the engine may need to pull the other. An OCO pair needs its sibling cancelled. This must happen **between fills of one sweep**, not after `process()` returns — otherwise the very next fill, or a stop the fill triggers, could trade against a leg the engine is about to report as cancelled (findings **X3**, **X15**). The callbacks run synchronously inside the sweep; the engine's handler cancels the sibling through `book.cancel_order()`, which invalidates its entry, and the next `_peek` skips it. They are skipped in an auction (`both_resting`), whose follow-ups the engine runs after `execute_uncross`.

6.15.4 The Three Last Prices

Field	Updated when	Used for
`last_trade_price`	every trade, continuous and auction	stop and trailing-stop triggers, depth window centre, price collars, `run_stops` after an uncross
`last_buy_price`	a trade whose **aggressor was the buyer**	market data ("last trade on the buy side"), persisted across restarts
`last_sell_price`	a trade whose **aggressor was the seller**	the same, for the sell side

```
if not both_resting:
    if aggressor.side == Side.BUY:
        self.last_buy_price = fill_price
    else:
        self.last_sell_price = fill_price
```

An aggressive **buy** (lifting the offer) updates `last_buy_price`; an aggressive **sell** (hitting the bid) updates `last_sell_price`. Auction prints have no aggressor (`aggressor_side == "AUCTION"`, finding **L9**), so they update only `last_trade_price`. Consumers of `Trade.aggressor_side` must therefore expect a third value besides `"BUY"` and `"SELL"`. After an engine restart `restore_stats()` puts the two side-specific prices back.

6.16 Worked Example: One LIMIT Order, Step by Step

This is real output from the script in Learning by Debugging. Prices are ticks with two decimals, and order ids are shortened.

Starting book. Three asks and one bid have been rested:

Order	Side	Qty	Price	Gateway	`arrival_seq`
a1	SELL	300	100.10	GW1	1
a2	SELL	200	100.10	GW2	2
a3	SELL	500	100.20	GW2	3
b1	BUY	400	100.00	GW3	4

```
_asks heap       : [((10010, 1), a1), ((10010, 2), a2), ((10020, 3), a3)]
_bids heap       : [((-10000, 4), b1)]
_ask_qty         : {10010: 500, 10020: 500}
_bid_qty         : {10000: 400}
_orders_by_gw    : {GW1: [a1], GW2: [a2, a3], GW3: [b1]}
```

Incoming: BUY LIMIT 400 @ 100.10 from GW3.

```
           ASKS                                  incoming
  100.20 | a3 500                                BUY 400 @ 100.10
  100.10 | a1 300  a2 200    <— best ask          (crosses the spread)
  -------+------------------
  100.00 | b1 400            <— best bid
           BIDS
```

1. `process` → `_match_limit` → `_sweep(_asks, price_limit=10010)`.
2. Pass 1: `_peek` → **a1** (key `(10010, 1)` is the heap top). 100.10 100.10, different gateway, `fill_qty = min(400, 300) = 300`. `_apply_fill` creates trade **300 @ 100.10**. The aggressor becomes PARTIAL (100 left); a1 becomes FILLED and is purged (entry invalid, removed from all three id-keyed indexes). `_ask_qty[10010]` drops 500 → 200.
3. Pass 2: `_peek` pops a1's dead entry and returns **a2**. `fill_qty = min(100, 200) = 100`. Trade **100 @ 100.10**. The aggressor is FILLED and purged (a no-op, it never rested); a2 is PARTIAL with 100 left. `_ask_qty[10010]` drops 200 → 100.

4. The loop ends (`remaining_qty == 0`), and `_match_limit` has nothing to rest.
5. `process`: there are trades, but `_has_stops` is `False`, so no cascade.

```
trades : [(10010, 300), (10010, 100)]
events : [(agg, FILLED, 0), (a1, FILLED, 0), (agg, FILLED, 0), (a2, PARTIAL, 100)]
_asks  : [((10010, 2), a2), ((10020, 3), a3)]
_ask_qty: {10010: 100, 10020: 500}
```

Notice that the aggressor appears **twice** in `events` and shows FILLED both times, although it was PARTIAL after the first fill: the list holds references to one mutable object. That is why the engine de-duplicates before publishing, and why tests that want the intermediate states look at the trades rather than the events.

6.17 Resting an Order: `_rest()`

`_rest` is the **only** way a LIMIT or ICEBERG order becomes visible on the book. Five structures are updated together; miss one and the book silently corrupts:

```
def _rest(self, order: Order) -> None:
    assert order.price_ticks is not None, "Resting order must have a price"
    price = order.price_ticks
    qty = (
        order.displayed_qty or 0
        if order.order_type == OrderType.ICEBERG
        else order.remaining_qty
    )
    order.arrival_seq = self._next_seq()                    # time priority (H1)
    self._note("RESTED", order.id, order.arrival_seq)       # trace record
    if order.side == Side.BUY:
        key = (-price, order.arrival_seq)
        heap = self._bids
        self._bid_qty[price] = self._bid_qty.get(price, 0) + qty   # 1. level index
    else:
        key = (price, order.arrival_seq)
        heap = self._asks
        self._ask_qty[price] = self._ask_qty.get(price, 0) + qty
    entry = _HeapEntry(key=key, order=order)
    heapq.heappush(heap, entry)                              # 2. heap
    self._order_index[order.id] = order                      # 3. id → Order
    self._entry_index[order.id] = entry                      # 4. id → entry
    self._track_gateway(order)                               # 5. gateway index (+ GTC count)
```

`price = order.price_ticks` binds the narrowed `int` to a local (finding **L7**): mypy forgets attribute narrowing after a method call, and the local avoids `type: ignore` comments in the arithmetic.

6.18 Cancelling: `cancel_order()`

```
order = self._order_index.get(order_id) or self._triggered.get(order_id)
if order is None:
    return None
if order.status in (FILLED, CANCELLED, REJECTED, EXPIRED):
    return None
order.status = OrderStatus.CANCELLED
entry = self._entry_index.get(order_id)
if entry:
    entry.valid = False
    o = entry.order
    if o.order_type not in (OrderType.STOP, OrderType.STOP_LIMIT) and o.price_ticks is not None:
        qty = o.displayed_qty or 0 if o.order_type == OrderType.ICEBERG else o.remaining_qty
        self._deduct_qty_index(o, qty)
self._purge_from_indexes(order)
return order
```

This is lazy deletion in action. The function can cancel three kinds of order:

- a resting LIMIT/ICEBERG — entry invalidated, visible quantity deducted;
- an armed STOP/STOP_LIMIT — entry invalidated, **no** level deduction (dormant stops are not in the level index);
- a trailing stop or a triggered stop waiting in `_triggered` — no entry at all; the status change is enough. `_check_trailing_stops` drops cancelled trailing stops on its next scan, and `process()` returns immediately for a cancelled triggered stop.

It returns `None` for unknown or already-terminal orders, which the engine turns into a cancel reject. Cost: O(1) dictionary work, plus the occasional amortised `_compact_heaps`.

6.19 Amending: `amend_order()`

`amend_order(order_id, new_price=None, new_qty=None, now=None)` returns `(order, priority_reset, reason)`. Only resting LIMIT and ICEBERG orders can be amended, and the new values are validated: quantity > 0, quantity greater than what has already been filled, and price > 0.

The priority rule matches real exchanges:

Change	Queue position
Quantity **decrease**, same price	**kept** — same heap entry, same key
Quantity increase	**lost** — new `arrival_seq`, back of the queue
Any price change	**lost** — new key at the new price

The order of the bookkeeping matters: first deduct the **old** visible quantity at the **old** price (`_deduct_qty_index` reads `order.price_ticks`), then mutate the order, then add the **new** visible quantity at the **new** price. Swapping these steps corrupts `_bid_qty`/`_ask_qty`.

```
if priority_reset:
    order.ts_ns = now
    order.arrival_seq = self._next_seq()
    if entry:
        entry.valid = False                    # retire the old entry
    if order.side == Side.BUY:
        key = (-price, order.arrival_seq)
        heap = self._bids
        self._bid_qty[price] = self._bid_qty.get(price, 0) + new_visible
    else:
        ...
    new_entry = _HeapEntry(key=key, order=order)
    heapq.heappush(heap, new_entry)
    self._entry_index[order.id] = new_entry
else:
    # same entry, same key: keeping your place in the queue
    ...  # only the level index gets the new visible amount
```

A heap key cannot be edited in place, so "moving" an order is "retire the old entry, push a new one". For an iceberg, a priority-losing amend refills the displayed slice to a full peak; a priority-keeping amend may only *shrink* the current slice — refilling it in place would jump the orders queued behind it.

Real output: two bids at 100.00, `x1` (seq 1) and `x2` (seq 2), then two amends of `x1`:

```
amend x1 qty 300 → 200  → priority_reset=False   _bids keys: (-10000,1)  (-10000,2)
amend x1 qty 200 → 400  → priority_reset=True    _bids keys: (-10000,1) valid=False
                                                             (-10000,2)
                                                             (-10000,3)  ← x1 now behind x2
```

Details that are easy to miss:

- **A priority-losing amend refills an iceberg's peak** (`displayed_qty = min(visible_qty, remaining_qty)`), because the order is re-queued at the back anyway. A priority-keeping amend (quantity decrease only) can merely *shrink* the current slice, `min(displayed_qty, remaining_qty)`.

- **`quantity` is the order's new total**, not the new remainder: the book keeps `filled_qty` and sets `remaining_qty = qty - filled_qty`, which is why a new quantity at or below the filled amount is refused.
- **A rejected amend changes nothing**: every validation runs before the first mutation, so a refusal never leaves the level index half-updated.
- **Only `LIMIT` and `ICEBERG` can be amended.** Stops, trailing stops and the like answer `Cannot amend <TYPE> orders`.
- **A priority loss is traced** as a `RESTED` record with the new `arrival_seq`, and the engine returns it as `priority_seq` in `order.amended`.

Amending into a crossing price

`amend_order` never matches. If an amend moves a bid up to or through the best ask, the engine (`Engine._handle_amend`) checks `book.best_ask_ticks()` / `book.best_bid_ticks()` and, if the order became marketable, cancels it and runs it through `process()` again as a fresh aggressive order. Otherwise the book would be left crossed (invariant I2; finding **H2**). This is skipped when matching is off (auction, halt), where a crossed book is normal.

The book is only the last step of an amend. `_handle_amend` first applies the protections a new order faces, so an amend cannot be used to dodge them: the gateway check, order lookup, **ownership** (`NOT_OWNER`), **tick size** (`TICK_VIOLATION`; the amend message carries display money, converted with `to_ticks_exact`), the **session** gate (`MARKET_CLOSED`), the **price collar** (`COLLAR_BREACH`) and the **order-size / notional limits** on the *amended* total quantity and price.

6.20 Icebergs

An iceberg shows only a slice (`displayed_qty`, at most `visible_qty`) of its total size. Three rules make it fair:

1. **A resting iceberg can be hit only for its displayed slice per pass** — the `_passive_avail` cap in `_sweep` (finding #17).
2. **When the slice is used up and hidden quantity remains, a new slice is shown at the back of its price level.** `_apply_fill` deducts the traded slice from the level index and `_reinsert_iceberg` retires the old entry, takes a fresh `arrival_seq`, pushes a new entry and adds the new peak to the level index.
3. **The level index only ever holds the displayed slice.**

```
def _reinsert_iceberg(self, order: Order) -> None:
    price = order.price_ticks
    old_entry = self._entry_index.get(order.id)
    if old_entry:
        old_entry.valid = False
    order.arrival_seq = self._next_seq()             # back of the queue
    new_peak = order.displayed_qty or 0
    if order.side == Side.BUY:
        key = (-price, order.arrival_seq)
        heap = self._bids
        self._bid_qty[price] = self._bid_qty.get(price, 0) + new_peak
    else:
        key = (price, order.arrival_seq)
        heap = self._asks
        self._ask_qty[price] = self._ask_qty.get(price, 0) + new_peak
    entry = _HeapEntry(key=key, order=order)
    heapq.heappush(heap, entry)
    self._entry_index[order.id] = entry
```

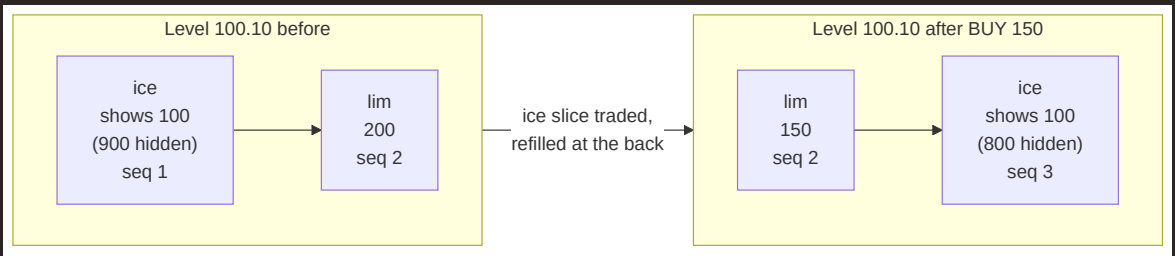
6.20.1 Worked example (real output)

An ICEBERG SELL of 1000 with a peak of 100 rests at 100.10 (GW1), then a plain LIMIT SELL of 200 at the same price (GW2). A BUY of 150 arrives (GW3).

```
before   _asks: [((10010,1), ice, valid), ((10010,2), lim, valid)]
         _ask_qty: {10010: 300}            ← 100 shown by ice + 200 lim

pass 1   _peek → ice; _passive_avail = displayed 100; fill 100 @ 100.10
         slice used up, 900 hidden left → new peak 100
         _deduct_qty_index(100): 300 → 200; _reinsert_iceberg: +100 → 300
         ice gets key (10010,3): now BEHIND lim
pass 2   _peek pops the dead (10010,1) entry → lim; fill 50 @ 100.10
         _ask_qty: 300 → 250

after    _asks: [((10010,2), lim 150, valid), ((10010,3), ice, valid)]
trades   [(ice, 10010, 100), (lim, 10010, 50)]
```



Had the sweep taken 150 from the iceberg in one go, the iceberg's hidden quantity would have traded ahead of `lim`, which was visibly waiting at that price before the hidden part existed for anyone. That was finding #17.

6.21 Self-Match Prevention (SMP)

In an auction uncross there is no aggressor; see Self-Match Prevention in an Auction Uncross.

An order from gateway G may not trade with a resting order from the same G (a *wash trade*). The policy travels on the order as `Order.smp_action`. If the client did not specify one, the engine resolves it to the gateway's configured default before the order reaches the book (`Engine._resolve_smp_action`), so the book can treat it as a concrete value. (It still writes `order.smp_action or SmpAction.NONE` defensively.)

`SmpAction`	Effect when the sweep meets a same-gateway resting order
`NONE`	Explicitly allowed: the trade happens.
`CANCEL_AGGRESSOR`	The incoming order is CANCELLED (`SELF_MATCH_PREVENTED`); the resting order stays. The sweep returns.
`CANCEL_RESTING`	The resting order is cancelled (`_smp_cancel_resting`) and the sweep **continues** with the next order.
`CANCEL_BOTH`	Both are cancelled; the sweep returns.

```
if _smp_action != SmpAction.NONE and _gw_id == best.gateway_id:
    if _smp_action == SmpAction.CANCEL_AGGRESSOR:
        aggressor.status = OrderStatus.CANCELLED
        aggressor.cancel_reason = "SELF_MATCH_PREVENTED"
        events.append(aggressor)
        return
    elif _smp_action == SmpAction.CANCEL_RESTING:
        self._smp_cancel_resting(best, events)
        continue  # skip this resting order; try next
    elif _smp_action == SmpAction.CANCEL_BOTH:
        self._smp_cancel_resting(best, events)
        aggressor.status = OrderStatus.CANCELLED
        aggressor.cancel_reason = "SELF_MATCH_PREVENTED"
        events.append(aggressor)
        return
```

A cancelled aggressor keeps its `remaining_qty`; that is exactly why the matchers test `status` before resting (C3). `_smp_cancel_resting` deducts the resting order's visible quantity, purges it through `_purge_from_indexes` like every other removal, appends it to `events` and reports a dead OCO leg. FOK orders resolve SMP up front in their pre-check (see FOK). `tests/test_engine_oco_smp.py` covers the interplay with OCO.

6.22 Stop Orders

A stop is **not in the book**. It is in no price level, never matches and appears in no depth until a trade price reaches its trigger. Then it becomes a real order:

Type	Fires when	Becomes
BUY STOP / STOP_LIMIT	`last_trade_price >= stop_price_ticks`	MARKET (`price_ticks=None`) / LIMIT at its limit price
SELL STOP / STOP_LIMIT	`last_trade_price <= stop_price_ticks`	MARKET / LIMIT
TRAILING_STOP	its moving trigger is reached	MARKET

6.22.1 Arming: `_add_stop`

```
order.arrival_seq = self._next_seq()     # ties at the same trigger (H1 / M10)
self._note("RESTED", order.id, order.arrival_seq)
if order.side == Side.BUY:
    key = (order.stop_price_ticks, order.arrival_seq)
    entry = _HeapEntry(key=key, order=order)
    heapq.heappush(self._buy_stops, entry)
else:
    key = (-order.stop_price_ticks, order.arrival_seq)
    entry = _HeapEntry(key=key, order=order)
    heapq.heappush(self._sell_stops, entry)
self._order_index[order.id] = order
self._entry_index[order.id] = entry
self._track_gateway(order)
self._has_stops = True
events.append(order)   # a NEW state notification, not a second ACK
```

If the stop is already breached on arrival (M2), `process()` calls `_trigger_stop_now` instead. It applies the same conversion (STOP and TRAILING_STOP → MARKET, STOP_LIMIT → LIMIT at its limit price), resets `ts_ns`, and matches the order straight away. Such an order was never armed, so it is in no index and never visits `_triggered`; a STOP_LIMIT that does not fill completely then rests through `_rest` and gets a fresh `arrival_seq` like any LIMIT. The check uses `last_trade_price` only: with no trade yet in the session, every stop is armed, however close its trigger is to the market.

A stop is also **not amendable**: `amend_order` accepts only resting LIMIT and ICEBERG orders and answers `Cannot amend STOP orders` (and likewise for STOP_LIMIT and TRAILING_STOP). Cancel and re-enter instead.

6.22.2 Triggering: `_check_stops`

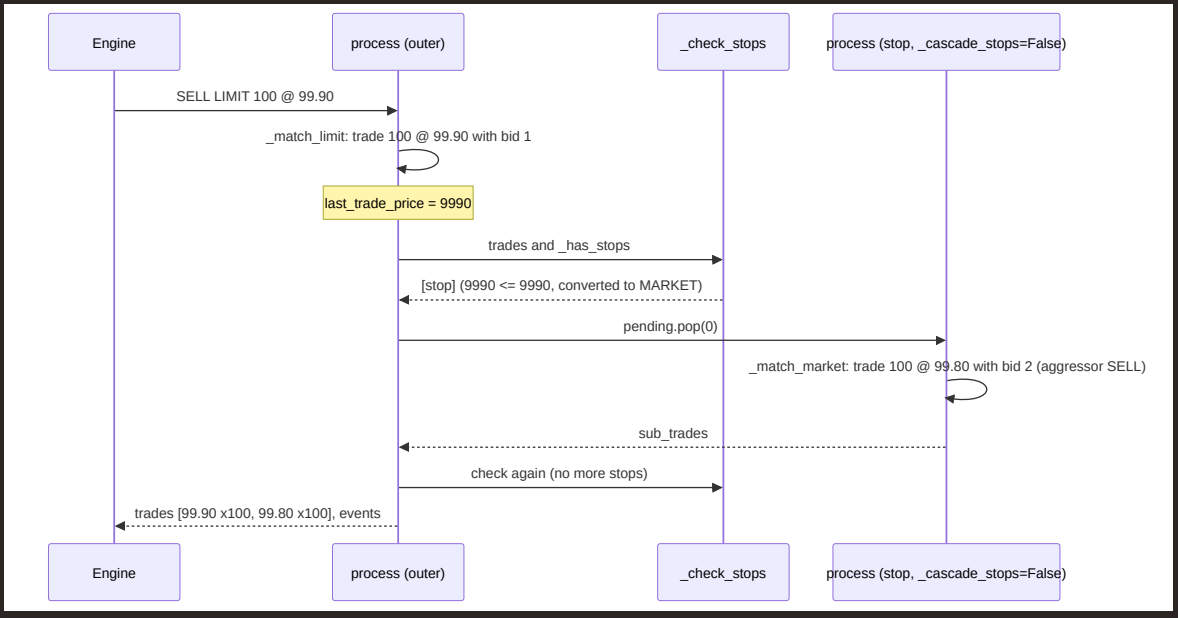
```
while self._buy_stops:
    entry = self._buy_stops[0]
    if not entry.valid or entry.order.status in (CANCELLED, EXPIRED):
        heapq.heappop(self._buy_stops)      # lazy deletion
        continue
    stop_price, _ = entry.key
    if self.last_trade_price < stop_price:
        break                               # lowest buy stop not reached: none are
    heapq.heappop(self._buy_stops)
    stop_order = entry.order
    self._note_trigger(stop_order)          # trace: armed as, on which trade, becomes
    if stop_order.order_type == OrderType.STOP:
        stop_order.order_type = OrderType.MARKET
        stop_order.price_ticks = None
    else:
        stop_order.order_type = OrderType.LIMIT
    stop_order.ts_ns = now
    self._order_index.pop(stop_order.id, None)
    self._entry_index.pop(stop_order.id, None)
    self._discard_from_gateway_index(stop_order)
    triggered.append(stop_order)
# ... the same for _sell_stops, with the key negated back ...
for stop_order in triggered:
    self._triggered[stop_order.id] = stop_order
return triggered
```

Things to notice:

- **`heap[0]` decides.** If the closest stop has not been reached, none has, so a trade that triggers nothing costs one comparison per heap.
- **Triggering does not match.** It converts the order, takes it out of every index and returns it to `process()`, which matches it as a new order in the cascade loop. A STOP_LIMIT that does not fill completely re-enters the book through `_rest` like any LIMIT.
- **The waiting room.** Between being released and being processed, the order sits in `_triggered`. It is out of all indexes but `cancel_order` can still find it there — an OCO sibling's fill may cancel it before its turn (X15).

6.22.3 Worked example: a stop cascade (real output)

Two bids rest (100 @ 99.90 and 100 @ 99.80, GW1), and a SELL STOP 100 with trigger 99.90 is armed (GW2) — sell-stop key `(-9990, 3)`. Then a SELL LIMIT 100 @ 99.90 arrives (GW3):




```
trades: [(seller=limit, 9990, 100, 'SELL'), (seller=stop, 9980, 100, 'SELL')]
stop now: MARKET FILLED
```

All trades from the cascade are returned in one `(trades, events)` pair, so the engine publishes the original order's fills and the stop's fills together.

6.23 Trailing Stops

A trailing stop's trigger follows the market at a fixed distance, `trail_offset_ticks`, and only ever **tightens**:

Side	Ratchet (after each trade at price p)	Fires when
SELL	`stop = max(stop, p - offset)` — moves **up** as the market rises	`p <= stop`
BUY	`stop = min(stop, p + offset)` — moves **down** as the market falls	`p >= stop`

Example from the source (SELL, offset 5): the market trades at 100 and the stop sits at 95. The market rises to 110, so the stop ratchets to 105. A fall back to 108 changes nothing, because the stop never moves down. A fall to 105 triggers it as a MARKET sell.

Because every trade can move every trailing trigger, they live in a plain list and `_check_trailing_stops` walks it all — O(t) per trade batch. It rebuilds the list (`still_active`) as it goes, which also lazily drops CANCELLED, EXPIRED and FILLED entries. A fired trailing stop becomes MARKET with `trail_offset_ticks = None`, is popped from `_order_index` and the gateway index, and goes to `_triggered` like any other stop.

Two corner cases follow from the plain list. A **cancelled** trailing stop is purged from the indexes at once, but stays in the list until the next `_check_trailing_stops` scan, and `_has_stops` stays true until that scan finds all three stop structures empty. And the check runs only after a trade (`last_trade_price` is its only input), so a quiet market never moves a trailing trigger.

The initial trigger is set by the engine, not the book: if the client sends no `STOP=`, `_handle_new_order` derives it as `last_trade_price - offset` (SELL) or `+ offset` (BUY), and rejects the order if there has been no trade yet. `_add_trailing_stop` asserts that both values are present.

6.24 OCO Orders and Quote Legs

Two engine-level concepts reach into the book only through the callbacks:

- **OCO (one-cancels-other).** Two orders share an `oco_group_id`; when one ends, the engine cancels the other. The book reports a dead leg through `on_oco_leg_done`, from `_apply_fill` (a leg FILLED), `_smp_cancel_resting`, and `_oco_leg_ended` (an aggressor that died in `process()`: an IOC remainder, an SMP or FOK refusal, a no-match reject, a triggered stop leg short of liquidity).
- **Quotes.** A market maker's two-sided quote is two ordinary resting orders with a `quote_id` and `origin = OrderOrigin.QUOTE`. `_apply_fill` calls `on_quote_leg_fill` for every fill of a quote leg so the engine can apply its refresh policy and pull the sibling leg straight away.

In both cases the engine's handler calls back into `book.cancel_order()`. That is safe in the middle of a sweep: the cancel only flips `valid`, adjusts the dicts and maybe compacts the heaps (compaction keeps the same list objects), and the next `_peek` simply skips the dead entry. See the guided tour, §10 for the engine side.

6.25 No-Matching Mode and Auctions

During an opening or closing auction, a reopening after a halt, or while a symbol is halted, the engine calls `process(order, match=False)`. Orders queue without trading, and the book may become crossed. When the call phase ends, `engine/auction.py` takes over:

1. `compute_equilibrium(book)` finds the single price that maximises the executable volume, reading the heaps (live entries only).
2. `execute_uncross(book, ...)` walks both heaps with `book._peek` and trades best bid against best ask at that one price, calling `book._apply_fill(..., both_resting=True)`.
3. `Engine._run_uncross` publishes the results and then calls `book.run_stops(now)`: the same stop cascade `process()` runs after a continuous trade (released stops queue FIFO behind the ones already waiting), so stops fire in the same order after an auction print.

`both_resting=True` changes three things in `_apply_fill`: both orders' visible quantity is deducted from the level index (both were resting), the trade's `aggressor_side` is `"AUCTION"`, and the quote/OCO callbacks are skipped (the engine runs those follow-ups after the uncross).

6.26 Self-Match Prevention in an Auction Uncross

Self-Match Prevention is easy to state for continuous trading: the incoming order is the *aggressor*, it meets one resting order at a time, and the policy says which of the two dies. An auction has no aggressor and no arrival order. Every order is already resting, the book is allowed to be crossed, and one price will be chosen for all of them. This section explains what the engine does about a gateway that is on both sides of that crossed book, why that is harder than it looks, and how the code does it.

6.26.1 What the SMP walk is for

Two requirements pull in opposite directions:

1. **No wash trade.** A bid and an ask of one gateway must not trade with each other, in an auction either. The gateway's `smp_action` decides which order is cancelled instead.
2. **The price must be honest.** The auction prints the price that executes the most volume. If a pair that will be cancelled still counts as volume when the price is chosen, the exchange publishes (and prints) a price that is based on trades that never happen.

The textbook equilibrium uses two *curves*: demand at price P is every bid at or above P, supply is every ask at or below P, and the executable quantity is `min(demand, supply)`. That arithmetic ignores who owns the orders. With SMP the quantity that can really trade at P is whatever is left after the pairs of one gateway have been removed, and that depends on *which* orders meet *which*, which the curves do not know. The **SMP walk** is the answer: instead of adding up the curves, act out the uncross at P order by order, apply the SMP rules where a gateway meets itself, and count what trades.

The same walk is used twice, in two modes:

Mode	Function	Mutates the book?	Used for
Dry run	`_simulated_quantity(book, P)`	No: works on copies (`_CandidateOrder`)	Choosing the price: `compute_equilibrium`, hence the indicative published every interval during a call phase
Execution	`execute_uncross(book, P)`	Yes: real fills, real cancels	The uncross itself, when the phase ends, a halt reopens, or an operator runs `reopen`

The two loops follow the same rules by construction and a randomised test (`test_indicative_matches_random_multi_gateway_uncross`) checks that the dry-run quantity equals what the execution trades.

6.26.2 The rules, exactly

At a candidate price P the walk takes the **candidate orders**: bids priced at P or above and asks priced at P or below, each side in price-time priority (best price first, then arrival order; an iceberg counts its hidden quantity but trades only its shown slice). It repeatedly looks at the head of each side, the best bid and the best ask:

1. **Same OCO group?** If the two are the two legs of one OCO pair, the pair can only be left standing as one order: the bid is cancelled and its sibling, the ask, ends with it (no SMP reason is recorded, since the gateway may well allow self-trades).
2. **Same gateway with a policy?** Both orders are resting, so neither is the aggressor. The convention (`_auction_smp_action`) is: **the bid is treated as the aggressor and the ask as the resting order.** The bid's policy applies; if the bid's policy is `NONE`, the ask's policy applies, with the ask as the aggressor. Then:

Policy	Cancelled	Survives and meets the next order
`CANCEL_AGGRESSOR`	the aggressor (the bid; the ask only when the ask's policy is the one used)	the passive order
`CANCEL_RESTING`	the passive order	the aggressor
`CANCEL_BOTH`	both	neither: the walk moves on to the next pair
`NONE` on both	nobody: the pair is free to trade (step 3)	

3. **Otherwise they trade.** `fill = min(shown bid, shown ask)`, both reduce, a finished order leaves its level, a used-up iceberg slice refills at the back of its level.

The loop ends when one side has no candidate left. A cancel in step 1 or 2 takes the whole order away, an iceberg's hidden reserve included, as one cancel.

The dry run adds up `traded`, and the totals needed for the indicative: *buy total* = traded + what survives of the candidate bids, *sell total* likewise; the surplus is `|buy total - sell total|` and the imbalance side is the larger total's. Cancelled orders count as zero, so the imbalance the engine publishes is the imbalance **after** SMP.

6.26.3 How the price is chosen with the walk

`compute_equilibrium` still starts from the curves, because they give a cheap **upper bound**: SMP can only remove volume, so the walk at P can never exceed `min(demand(P), supply(P))`. The steps are:

1. Build the curves in one pass each (O(P log P)).
2. `_needs_simulation(book)`: if no gateway has an SMP-capable bid at or above one of its own asks (and no OCO pair has both legs in the book), the curves are exact and the walk is skipped. This is the usual case, and the reason the indicative stays cheap.

3. Otherwise try the candidate prices **best bound first**. Walk each one; stop as soon as the next bound is smaller than the best walk found, because nothing after it can win.
4. The winner is the price with the most executed quantity, then the smallest surplus, then the lowest price (the same tie-break as without SMP).

6.26.4 Executing the uncross

`Engine._run_uncross` (end of phase, halt reopen, circuit-breaker reopen, operator `reopen`) does, per symbol:

```
result = compute_equilibrium(book)
price  = result.eq_price, or result.resolve_price when nothing prints
trades, events = execute_uncross(book, price)      # real fills and cancels
publish trades and events; run the stops the print triggers; publish auction.result
```

`resolve_price` exists for the case where SMP (or an OCO pair) consumes the whole overlap: the book is crossed, **nothing trades**, but the offending orders must still be cancelled. The price is used to run the walk; it is never published as an equilibrium (`eq_price` stays `None`). An operator- or breaker-asserted price (`price_override`) replaces the discovered price and the walk is run at that price instead.

`execute_uncross(book, P)` then does the walk of the rules above for real, and **ends with a resolve pass** (next section).

6.26.5 Why a resolve pass is needed: the pair the walk never meets

The walk at P only sees orders that can trade *at P*. A bid at 101.00 and an ask at 100.50 of the same gateway, with P = 100.00, are both outside it: the ask is above P and so is not a candidate, and the bid trades with whatever is at or below 100.00 and then has quantity left. After the uncross the book holds a bid **above** an ask: a crossed book, and in the continuous session that follows it is exactly the situation SMP exists to prevent.

This is not a rare corner. Two prices can execute the same volume once SMP has cancelled something, and the tie-break (least surplus, then lowest price) may pick the one at which the self-matching pair is *not* in range. It was seen in the full-day simulation of 2026-11-04: five symbols opened with crossed or locked books.

The resolve pass is the fix. After the walk, while best bid >= best ask:

- if the pair is an OCO pair, or a same-gateway pair with an SMP policy, apply the same rules as in the walk (cancel, **no trade**) and look again;
- otherwise stop. A crossed pair that is free to trade would have been found by the equilibrium, so it cannot be left; nothing further is done.

The pass cancels only and never prints, so it cannot change the equilibrium price, the executed quantity or the indicative. It does change the book, and the cancels are published (`order.cancelled`, reason `SELF_MATCH_PREVENTED`) in the same dispatch as the print, followed by the normal book snapshot.

6.26.6 A full walk-through (no price conflict)

Three orders, no other interest. `GW1` has `CANCEL_AGGRESSOR`; the numbers are prices and quantities:

Order	Side	Gateway	Price	Qty	SMP
B	BUY	GW1	101.00	20	`CANCEL_AGGRESSOR`
SA	SELL	GW1	100.50	10	`NONE`
SB	SELL	GW2	100.00	10	`NONE`

The curves say 20 at 100.50 or 101.00 (demand 20, supply 20) and 10 at 100.00.

The walk at each candidate price:

Price	Curve	Walk	Why
100.00	10	**10**	Candidates: B, SB. B trades SB for 10. B has 10 left, no ask: end. Surplus 10 (BUY side).
100.50	20	**10**	Candidates: B, SB, SA. B trades SB for 10. B (10 left) meets SA: same gateway, B's policy `CANCEL_AGGRESSOR`: B is cancelled. Traded 10. Surplus 10.
101.00	20	**10**	Same as 100.50.

Result: 10 executed at all three prices, surplus 10 at all three, so the **lowest price, 100.00**, wins. `eq_price = 100.00`, `eq_qty = 10`.

Execution at 100.00: B trades SB for 10 at 100.00 (SB is filled). The walk ends because SA (100.50) is above P. Book now: B 10 @ 101.00 against SA 10 @ 100.50: **crossed**. The resolve pass compares B and SA: same gateway, B's policy `CANCEL_AGGRESSOR`, so B is cancelled (`SELF_MATCH_PREVENTED`, 10 left). SA stays. The book opens with one ask, uncrossed.

(This is `test_uncross_leaves_no_self_match_cross_above_the_equilibrium` in `tests/test_auction_oco_smp.py`.)

6.26.7 A harder walk-through: the curves overstate by a factor of ten

This one is taken, simplified, from a real opening in the simulation. Four gateways; GW1 and GW4 are on both sides.

Order	Side	Gateway	Price	Qty	SMP
b1	BUY	GW3	101.00	10	`NONE`
b2	BUY	GW1	101.00	40	`CANCEL_AGGRESSOR`
b3	BUY	GW2	100.50	50	`CANCEL_RESTING`
b4	BUY	GW4	100.50	500	`CANCEL_AGGRESSOR`
a1	SELL	GW1	100.00	30	`NONE`
a2	SELL	GW4	100.20	700	`NONE`

Per price:

Price	Curve `min(demand, supply)`	Walk	Surplus after SMP
100.00	30	30	530
100.20	**600**	**60**	670
100.50	600	60	670
101.00	50	10	720

The curves would print 600 at 100.20. The walk at 100.20 (candidates: b1, b2, b3, b4 against a1, a2):

1. b1 (GW3) vs a1 (GW1): different gateways, **trade 10**. a1 has 20 left.
2. b2 (GW1) vs a1 (GW1): same gateway, b2's `CANCEL_AGGRESSOR`: **b2 is cancelled** (40 gone). a1 stays.
3. b3 (GW2) vs a1 (GW1): **trade 20**. a1 is done. b3 has 30 left.
4. b3 (GW2) vs a2 (GW4): **trade 30**. b3 is done. a2 has 670 left.
5. b4 (GW4) vs a2 (GW4): same gateway, b4's `CANCEL_AGGRESSOR`: **b4 is cancelled** (500 gone). No bids remain: end.

Traded 10 + 20 + 30 = **60**. 100.20 and 100.50 both give 60 with surplus 670, so the lower price, **100.20**, is chosen. 540 of the 600 shares the curves counted belonged to one gateway trading against itself (b4 vs a2) or were cancelled before they could meet anyone. Without the walk the indicative would have advertised 600 and the print would have been 60.

6.26.8 Complications this adds

Complication	What goes wrong without care	How it is handled
No aggressor.	`CANCEL_AGGRESSOR` and `CANCEL_RESTING` have no natural meaning.	A convention: the bid is the aggressor, unless its policy is `NONE`, then the ask's policy applies with the ask as aggressor. It is documented for participants, so one gateway's outcome is predictable from its own policy.
Volume is not a curve.	`min(demand, supply)` overstates; the quantity at P depends on who meets whom.	The walk replaces the formula; the curves remain as an upper bound for pruning.
Cost.	A walk per candidate price on every indicative interval, for every symbol, is O(prices x orders).	`_needs_simulation` skips the walk when SMP cannot bite; the bound lets most candidates be skipped; the candidate prefix is sorted once per calculation (`_live_orders`).
Order dependence.	Which order is cancelled depends on priority: an earlier same-gateway pair in the queue shields a later one.	The walk follows price-time priority exactly as execution does; a cancelled order is flagged dead and dropped when it reaches the head of its level, preserving the queue.
Cancels change the volume of others.	Cancelling b2 above changed what a1 could trade with b3.	Within the walk, a cancel is applied immediately; a cancelled order's OCO sibling is killed in the same walk.
Equal-volume prices leave different books.	The tie-break can choose a price where a self-matching pair is out of range and stays crossed.	The resolve pass after the walk.
Nothing trades but orders die.	A book where SMP consumes the whole overlap has `eq_qty = 0`, so a naive "no equilibrium: skip" would leave it crossed.	`resolve_price`: the uncross still runs, to cancel; `auction.result` carries `eq_price: null`.
Imbalance is post-SMP.	Publishing an imbalance that counts doomed orders misleads participants.	Totals are taken from the survivors.
Icebergs.	The hidden reserve counts toward volume but trades only slice by slice, and a SMP cancel kills the reserve too.	Volume uses the full quantity; a fill is the shown slice, refilling at the back of the level; a cancel removes the whole order.
Overridden price.	An operator or breaker can assert a price that the walk did not pick.	The walk and the resolve pass run at that price; the reported `eq_qty` is what actually executed.
Verification.	An independent check has to model the same rules or it reports every SMP auction as wrong.	The simulation forensics replay (`tools/sim_forensics/checks/matching.py`, `_walk`) is a second, independent implementation of the walk; D6 and D10 compare the engine's prints and indicatives with it, and a randomised test checks the replay against `compute_equilibrium`.

6.26.9 Where this lives in the code

Piece	Location
Dry-run walk	`auction._simulated_quantity` (`_candidate_levels`, `_head`)
When the walk is needed	`auction._needs_simulation`
Price selection	`auction.compute_equilibrium`
Policy rule for a resting pair	`auction._auction_smp_action`, `order_book._self_match_action`
One pair, shared by walk and resolve pass	`auction._resolve_self_match`
Execution and resolve pass	`auction.execute_uncross`
Caller	`Engine._run_uncross`
Participant-facing description	Operator guide, *Risk controls*, SMP

6.27 Trace Records: What the Book Reports Besides Trades

Trades and order events do not show everything that happens inside a book. An order that *rests* gets a queue position nobody is told about; an iceberg refill re-queues the order silently; a trailing trigger moves without a trade; a stop turns into another order type. The book therefore appends small tuples to `self.trace` through `_note(kind, *fields)` (and `_note_trigger` for stops). `_note` does nothing when `trace` is `None`, so unit tests pay nothing.

Record	Written by	Fields after `(kind, symbol, ...)`
`RESTED`	`_rest`, `_add_stop`, `_add_trailing_stop`, `amend_order` (priority lost)	`order_id`, `arrival_seq` (`None` for a trailing stop, which has no sequence)
`ICEBERG_REFILL`	`_reinsert_iceberg`	`order_id`, new `arrival_seq`, new peak
`TRAILING_RATCHET`	`_check_trailing_stops`	`order_id`, new stop price, trade price
`TRIGGERED`	`_note_trigger`, from `_check_stops`, `_check_trailing_stops` and `_trigger_stop_now`	the `Order`, armed type, stop price, `last_trade_id`, `last_trade_price`, what it `becomes` (`MARKET`/`LIMIT`)
`SMP_DECISION`	`_sweep`, the FOK pre-check in `_match_fok` (one per same-gateway order in range), `auction._resolve_self_match`	`order_id` whose policy applied, the other side's `order_id`, the policy

`TRIGGERED` is written **before** the order is converted, so it still holds the type it was armed as. The engine owns one list for all books (`Engine._trace`) and `_flush_trace` publishes it after each dispatch: `TRIGGERED` becomes `order.triggered.<GW>` for the owner, the others become `engine.trace.<SYM>` (audit only). The book never publishes; it only reports.

Two more `engine.trace` kinds say why the engine decided, and the engine publishes them itself, outside the book's list: `UNCROSS_PRICE` from `Engine._run_uncross` just before the uncross runs (the price, its quantity and surplus, and the `rule` that chose it – most volume, then least surplus, then lowest price, or `OVERRIDE` for an imposed price – with the prices still tied when it did, from `AuctionResult.rule` / `tied_prices`), and `AON_PREFLIGHT` from `Engine._accept_combo`, one per leg of an all-or-none combo (what the leg could fill with and without the gateway's own orders, and whether the combo matches, rests or is rejected). See the message reference for the wire format.

6.28 Read-Only Views

Method	Returns	Cost	Used by
`get_order(id)`	resting `Order` or `None`	O(1)	engine cancel/amend routing

Method	Returns	Cost	Used by
`resting_orders()`	every NEW/PARTIAL order, stops included	O(resting)	persistence (`gtc_orders.json`), TIF expiry, mass actions
`orders_for_gateway(gw)`	that gateway's resting orders	O(k)	disconnect, kill switch, cancel-by-symbol
`quote_orders_for_gateway(gw)`	its QUOTE-origin legs	O(k)	quote reissue clean-up
`resting_count_by_gateway()`	`(gw, count)` pairs	O(gateways)	metrics
`best_bid_ticks()` / `best_ask_ticks()`	best price or `None`	O(levels)	crossing-amend check
`fillable_quantity(side, limit)`	executable qty incl. hidden	O(n)	combo AON pre-check (M5)
`book_stats()`	`BookStats`	O(levels + stops)	health beacon, metrics
`checkpoint()`	whole book as plain dicts: bids, asks and stops in priority order, last prices	O(n log n)	`system.book_checkpoint`, replay and restart
`restore_stats(...)`	— (sets `last_buy_price`, `last_sell_price`, and `last_trade_price` if still unset)	O(1)	startup restore
`containers()`	every container	—	memory size estimate
`depth_snapshot(tol)`	depth, imbalance, microprice, cost to move	O(levels)	market data
`snapshot()`	full aggregated book + last 5 trades	O(heap entries)	market data / viewer

`BookStats` holds `resting_orders`, `gtc_orders`, `stop_orders`, `bid_levels`, `ask_levels`, `heap_entries` (dead ones included), `live_entries` and `recent_trades`; its `heap_bloat_ratio` property is the heap size over the size `_compact_heaps` tolerates.

`checkpoint()` sorts each heap by key, so `bids`/`asks` come out in true matching priority with each order's `priority_seq`. Trailing stops carry no sequence (their `priority_seq` is `None`) and are listed in checking order.

Priority across a restart. `arrival_seq` is per process, but queue order must survive a restart. The persisted file lists orders by first arrival, which is wrong for an iceberg that was refilled or an order whose amend lost priority (finding **X26**), so `Engine._restore_gtc` sorts each book's orders by their saved `arrival_seq` before resting them. They receive fresh numbers, in the same relative order.

`snapshot()` walks the bid and ask heaps, skips dead entries, aggregates live orders by price level (an iceberg counts only its `displayed_qty`) and converts prices to display money with `from_ticks`. It includes `tick_decimals`, a monotonic `ts_ns`, the three last prices and only the **last 5** of the (by default) 20 `recent_trades`. `depth_snapshot()` reads only `_bid_qty`/`_ask_qty` within `tolerance_ticks` of `last_trade_price` and returns an empty dict before the first trade.

Neither is called per trade. The engine marks a symbol *dirty* after a change, and `Engine._flush_snapshots` publishes `snapshot()` and `depth_snapshot()` for dirty symbols at most once per `snapshot_interval_sec`. This keeps the O(n) snapshot cost out of the matching path.

6.29 Hot-Path Risk Checks Around the Book

The `OrderBook` is not the first line of defence. `Engine._handle_new_order` runs a sequence of checks *before* an order reaches `book.process()`, and `_publish_trade` applies a risk check *after* every trade. The true hot path is:

1. **Ingress validation and risk gates** (`_handle_new_order`)
2. **Matching** (`book.process` → `_sweep`)
3. **Post-trade risk reaction** (`_publish_trade` → `_check_circuit_breaker`)



Check	Where	Why it must run synchronously
Gateway allowed, connected and authenticated	`_handle_new_order` (fast path `_connected_participants`, fallback `_gateway_status`)	A disconnected or unauthorised participant must not inject live orders; rejecting early saves matching work.
Symbol allow-list	`_allowed_symbols`	Prevents trading in unconfigured instruments.
Field validation (M7)	`_validate_new_order`	Positive quantity and price, `visible_qty` in `(0, quantity]`, and the per-symbol order-size / notional caps. Bad values must be refused with a reason *before* the ACK, not crash inside the book.
Session state	`accepts_orders(self._session_state)`; ATO only in `OPENING_AUCTION`, ATC only in `CLOSING_AUCTION`	A wrongly accepted order cannot be "un-traded".
Halt gate	`_halted_symbols`	During a halt MARKET/IOC/FOK are rejected; LIMIT/ICEBERG may rest with matching off.
Price collar	`validate_collar(...)`	Anti-fat-finger; a bad print cannot be undone after matching.
Matching mode	`do_match = is_matching_enabled(self._session_state)`	Selects continuous matching or queue-only mode in `process()`. A halted symbol forces `do_match = False`.
No-match phase	`_handle_new_order`, step [9]	MARKET / IOC / FOK cannot wait in a queue, so they are refused with `SESSION_NOT_PERMITTED`; the book's own `match=False` rejection is a second line of defence.
Circuit breaker	`_publish_trade` → `_check_circuit_breaker(symbol, price, ts_ns)`	Must see each trade price in order, so the halt lands at the intended threshold.

ACK timing. Normally the engine publishes the positive ACK *before* calling `process()`. The exception is an order the book may still reject — any FOK, and MARKET/IOC/FOK while matching is off (`_book_may_reject`). Its ACK is sent only after the book has decided, so a rejected order gets a single `accepted=false` instead of `accepted=true` followed by a reject (finding **X5**).

Mass cancels. `risk.kill_switch` (`_handle_kill_switch`) and gateway disconnects (`_handle_gateway_disconnect`) must remove a participant's exposure fast. Quotes are pulled through the engine's quote index (`_quote_index.cancel_all_for_gateway`), and resting orders through `book.orders_for_gateway(gateway_id)` followed by `book.cancel_order()` for each. This is where `_orders_by_gateway` earns its place: the sweep is O(k) per book in the gateway's *own* orders, not O(n) over every order of every gateway. On a single-threaded engine, a full scan on every disconnect would freeze trading for everyone else while it ran — turning an emergency brake into an outage.

These checks cost CPU on every order, but there is no correct alternative: deferring them until after matching would turn hard pre-trade guarantees into best-effort clean-up after the damage is done.

6.30 Learning by Debugging

Reading is good; watching one order move through the code is better. Because `OrderBook` needs no network, you can drive it from a ten-line script and step through every function in this chapter.

6.30.1 1. A book lab script

Save this as `book_lab.py` in the **repository root** (do not commit it) and run it with `poetry run python book_lab.py`:

```
"""book_lab.py -- drive one OrderBook by hand and look inside it."""

import logging

from edumatcher.engine.order_book import OrderBook
from edumatcher.models.order import Order, OrderType, Side, SmpAction
from edumatcher.models.trade import set_run_seq

logging.basicConfig(level=logging.DEBUG, format="%(name)s: %(message)s")
set_run_seq(0)  # Trade ids need a run sequence; the engine sets it at startup.
book = OrderBook("DEMO")


def order(side, qty, price=None, gw="GW1", otype=OrderType.LIMIT, **kw):
    return Order.create(
        symbol="DEMO", side=side, order_type=otype, quantity=qty,
        gateway_id=gw, price_ticks=price, smp_action=SmpAction.NONE,
        tick_decimals=2, **kw,
    )


def show(label):
    print(f"--- {label}")
    for name in ("_asks", "_bids"):
        heap = getattr(book, name)
        print(f"  {name:6}:", [(e.key, e.order.id[:4], e.valid) for e in heap])
    print("  _ask_qty:", book._ask_qty, " _bid_qty:", book._bid_qty)
    print("  by gw  :", {g: [i[:4] for i in ids]
                         for g, ids in book._orders_by_gateway.items()})


a1 = order(Side.SELL, 300, 10010, gw="GW1"); book.process(a1)
a2 = order(Side.SELL, 200, 10010, gw="GW2"); book.process(a2)
a3 = order(Side.SELL, 500, 10020, gw="GW2"); book.process(a3)
b1 = order(Side.BUY, 400, 10000, gw="GW3"); book.process(b1)
show("starting book")

buy = order(Side.BUY, 400, 10010, gw="GW3")
trades, events = book.process(buy)  # <-- set a breakpoint here, then Step Into
print("trades:", [(t.price, t.quantity, t.aggressor_side) for t in trades])
print("events:", [(o.id[:4], o.status.value, o.remaining_qty) for o in events])
show("after BUY 400 @ 100.10")

book.cancel_order(a3.id)
show("after cancel a3")
```

Two details make a standalone script work: `set_run_seq(0)` (without it `Trade.create` raises "trade id requested before set_run_seq()"), and `tick_decimals=2` (the symbol `DEMO` is not registered in any config). Order ids are random, so yours will differ. The output (ids shortened, log lines abbreviated):

```
edumatcher.engine.order_book: LIMIT order resting after sweep ... side=SELL price=10010 remaining_qty=300
...
--- starting book
  _asks : [((10010, 1), 'e12b', True), ((10010, 2), '3ed6', True), ((10020, 3), 'eddb', True)]
  _bids : [((-10000, 4), '0661', True)]
  _ask_qty: {10010: 500, 10020: 500}  _bid_qty: {10000: 400}
  by gw  : {'GW1': ['e12b'], 'GW2': ['3ed6', 'eddb'], 'GW3': ['0661']}
edumatcher.engine.order_book: sweep fill ... passive_id=e12b... side=BUY price=10010 qty=300 aggressor_remaining_after=100
edumatcher.engine.order_book: sweep fill ... passive_id=3ed6... side=BUY price=10010 qty=100 aggressor_remaining_after=0
trades: [(10010, 300, 'BUY'), (10010, 100, 'BUY')]
events: [('b1cf', 'FILLED', 0), ('e12b', 'FILLED', 0), ('b1cf', 'FILLED', 0), ('3ed6', 'PARTIAL', 100)]
--- after BUY 400 @ 100.10
  _asks : [((10010, 2), '3ed6', True), ((10020, 3), 'eddb', True)]
  ...
--- after cancel a3
  _asks : [((10010, 2), '3ed6', True), ((10020, 3), 'eddb', False)]
  _ask_qty: {10010: 100}  _bid_qty: {10000: 400}
  by gw  : {'GW2': ['3ed6'], 'GW3': ['0661']}
```

Every worked example in this chapter came from a variation of this script. Change the orders and predict the output *before* you run it.

6.30.2 2. Debugger configurations

The repository does not ship a `.vscode/launch.json`. Create one with these three configurations (you need the Python and Python Debugger extensions):

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Book lab (current file)",
      "type": "debugpy",
      "request": "launch",
      "program": "${file}",
      "cwd": "${workspaceFolder}",
      "console": "integratedTerminal",
      "justMyCode": true
    },
    {
      "name": "One order-book test",
      "type": "debugpy",
      "request": "launch",
      "module": "pytest",
      "args": [
        "tests/test_order_flow.py::test_stop_order_triggers_into_market_after_trigger_trade",
        "--no-cov", "-q"
      ],
      "cwd": "${workspaceFolder}",
      "justMyCode": true
    },
    {
      "name": "pm-engine (DEBUG log to stdout)",
      "type": "debugpy",
      "request": "launch",
      "module": "edumatcher.engine.main",
      "args": ["-vv", "--log-target", "stdout"],
      "cwd": "${workspaceFolder}",
      "justMyCode": true
    }
  ]
}
```

- **Book lab** — open `book_lab.py`, set a breakpoint on the `book.process(buy)` line, press `F5`, then *Step Into* (`F11`). You land in `process()` and can follow the order all the way into `_sweep` and `_apply_fill`.
- **One test** — the test suite is the best collection of ready-made scenarios. `--no-cov` turns off coverage collection (part of the default `addopts`), which otherwise slows stepping. You can also use *Debug Test* in the VS Code Testing panel, or from a terminal: `poetry run pytest tests/test_order_flow.py -k stop --no-cov --pdb` (or put `breakpoint()` in a test).
- **The real engine** — start the engine under the debugger, start the other processes as usual, and send an order from `pm-alf-console`. Set a breakpoint in `Engine._handle_new_order` to see the risk gates, then step into `book.process`. The engine waits while you are stopped, so gateways may report missed heartbeats — that is expected when debugging a live system.

6.30.3 3. Where to put breakpoints

Set them by function name and follow the order as it moves down this table:

Breakpoint	Hits when	Inspect
`process` — first line	every order, including triggered stops	`order.order_type`, `match`, `self._triggered`
`_match_limit` (or another `_match_*`)	the matcher is chosen	`opposite is self._asks`
`_sweep` — `best = _peek(opposite_heap)`	every pass of the loop	`aggressor.remaining_qty`
`_sweep` — `_apply_fill(...)` call	just before each trade	`best.id`, `best_price`, `fill_qty`, `_passive_avail`
`_apply_fill` — `trades.append(trade)`	each trade is created	`trade`, `buy_order`, `sell_order`
`_apply_fill` — passive iceberg refill branch	an iceberg slice is used up	`passive.displayed_qty`, `new_peak`
`_purge_from_indexes`	an order leaves the book	`order.status`, `order.id in self._order_index`
`_rest`	a remainder starts resting	`key`, `qty`
`_compact_heaps` — first `for heap in heaps`	a compaction actually runs	`len(self._entry_index)`
`run_stops` — `t_order = pending.pop(0)`	a stop cascade step	`pending`, `t_order.order_type`
`_check_stops` — `triggered.append(stop_order)`	a stop fires	`stop_price`, `self.last_trade_price`
`cancel_order` / `amend_order`	a cancel / amend arrives	`entry.valid`, `priority_reset`

Conditional breakpoints (right-click a breakpoint → *Edit Condition*) stop only for the order you care about — for example `aggressor.id.startswith("b1cf")` in `_sweep`, or `order.order_type.value == "STOP"` in `process`. A **logpoint** prints without stopping, which is ideal inside the sweep loop: `fill {fill_qty} @ {best_price} vs {best.id[:4]}`.

6.30.4 4. Watch expressions that do not change the book

```
sorted((e.key, e.order.id[:4], e.valid) for e in self._asks)   # true priority order
self._ask_qty, self._bid_qty
{k[:4]: v.status.value for k, v in self._order_index.items()}
len(self._entry_index), len(self._bids) + len(self._asks)      # live vs. with garbage
[(t.price, t.quantity) for t in trades]
self.book_stats()
```

Some methods mutate the book — never call them from a watch

`self._peek(...)` pops dead entries, `self._check_stops(...)` and `self.run_stops(...)` release (and trade) stops, and `self._compact_heaps()` rebuilds heaps. Evaluating them in the debugger changes the state you are trying to observe. Read-only views (`snapshot()`, `depth_snapshot()`, `book_stats()`, `resting_orders()`, `get_order()`) are safe, and so is `sorted(...)` on a heap, which copies it.

From the debug console you can also check the whole book at any point:

```
from tests.engine_invariants import assert_book_invariants
assert_book_invariants(book, include_hygiene=True)   # use `self` inside OrderBook methods
```

If that raises, the message says which invariant (I1–I5) broke. This is the fastest way to find the exact step at which an edit to `order_book.py` breaks the synchronisation rules.

6.30.5 5. Reading the debug log

With `-vv` (or `--log-level DEBUG`) the engine's `edumatcher.engine.order_book` logger narrates matching without a debugger:

Log message	Emitted by
`sweep fill ... aggressor_remaining_after=`	`_sweep`, before each trade
`LIMIT order resting after sweep`	`_match_limit`
`MARKET order discarded` / `IOC cancelled`	`_match_market` / `_match_ioc`
`FOK rejected ... required= available=`	`_match_fok`
`SMP cancel aggressor` / `SMP cancel both` / `SMP cancel resting`	`_sweep`, `_smp_cancel_resting`
`stop armed`, `stop check`, `stop triggered`	`_add_stop`, `_check_stops`
`trailing stop ratchet` / `trailing stop triggered`	`_check_trailing_stops`
`cancel order_id=...`, `amend order_id=... priority_reset=`	`cancel_order`, `amend_order`

6.30.6 6. Experiments to try

1. **Price improvement.** Rest an ask at 100.00, send a BUY LIMIT at 101.00 and confirm the trade prints at 100.00.
2. **Iceberg fairness.** Rest an ICEBERG 1000/100 and a LIMIT 200 at the same price, send a BUY 150, and watch `_reinsert_iceberg` send the iceberg behind the LIMIT.
3. **SMP.** Give the aggressor the same `gw` as a resting order and try each `SmpAction`. Break in `_sweep` at the SMP branch.
4. **Stop cascade.** Rest bids at 99.90 and 99.80, arm a SELL STOP at 99.90, send a SELL 100 @ 99.90 and break at `t_order = pending.pop(0)` in `run_stops`.
5. **Lazy deletion and compaction.** Rest 200 asks far from the market, cancel them one by one and watch `len(book._asks)` grow garbage until `_compact_heaps` rebuilds the heap.
6. **FOK and hidden quantity.** Rest an ICEBERG 1000/100 and send a FOK BUY 500. `_ask_qty` shows only 100, yet the FOK fills — step through `_fok_available_qty` to see why.

6.30.7 7. Gotchas while debugging

- `len(book._bids)` includes dead entries; use `len(book._entry_index)` or `book_stats()`.
- A heap list is not sorted beyond `heap[0]`; wrap it in `sorted(...)` to read it in priority order.
- The same `Order` appears several times in `events`, always in its final state.
- A CANCELLED order still has `remaining_qty > 0`; test `status`.
- A triggered stop has changed type (STOP → MARKET), lost its price, and is in no index — look in `book._triggered`.
- `ts_ns` changes on refill, amend and trigger; queue position is `arrival_seq`.

6.31 Where to Look in the Code

All in `src/edumatcher/engine/order_book.py` unless stated otherwise:

You want to understand…	Read
The whole design in two pages	the module docstring
Price-time priority	`_HeapEntry`, `_rest`, `_next_seq`

You want to understand…	Read
The matching loop	`_sweep`
How a trade is recorded	`_apply_fill`
What happens to the unfilled part	`_match_market`, `_match_limit`, `_match_ioc`, `_match_fok`, `_match_iceberg`
Lazy deletion	`_peek`, `cancel_order`, `_purge_from_indexes`, `_compact_heaps`
Index maintenance	`_deduct_qty_index`, `_discard_from_gateway_index`, the comments in `__init__`
Icebergs	`_sweep` (the #17 cap), `_apply_fill` (refill), `_reinsert_iceberg`
Amends and priority	`amend_order`; the crossing-amend re-match in `Engine._handle_amend` (`engine/main.py`)
Stops	`_add_stop`, `_stop_is_triggered`, `_trigger_stop_now`, `_check_stops`, the cascade loop in `run_stops`
Trailing stops	`_add_trailing_stop`, `_check_trailing_stops`; initial trigger in `Engine._handle_new_order`
FOK	`_match_fok`, `_fok_available_qty`
Auctions	`engine/auction.py`: `compute_equilibrium`, `execute_uncross`; `Engine._run_uncross`
The engine side	`engine/main.py`: `_book`, `_handle_new_order`, `_publish_dispatch`, `_flush_snapshots`
What “correct” means	`tests/engine_invariants.py`, `tests/test_book_invariants_random_ops.py`, `tests/reference_matcher.py` (a deliberately naive matcher written from the documentation, used as an oracle)
Scenario tests	`tests/test_order_flow.py`, `tests/test_new_order_types.py`, `tests/test_amend.py`, `tests/test_engine_oco_smp.py`, `tests/test_order_book_compaction.py`, `tests/test_order_book_coverage.py`
Why the code is micro-optimised	`docs-design/perf-notes.md`

6.32 Data Structure Summary

Structure	Type	Key	Value	Purpose
`_bids`	heap (list)	`(-price_ticks, arrival_seq)`	`_HeapEntry`	Best bid, price-time order
`_asks`	heap (list)	`(price_ticks, arrival_seq)`	`_HeapEntry`	Best ask, price-time order
`_buy_stops`	heap (list)	`(stop_price_ticks, arrival_seq)`	`_HeapEntry`	Next buy stop to fire
`_sell_stops`	heap (list)	`(-stop_price_ticks, arrival_seq)`	`_HeapEntry`	Next sell stop to fire
`_trailing_stops`	list	—	`Order`	Trailing stops, re-evaluated after trades
`_triggered`	dict	`order_id`	`Order`	Fired stops waiting in the cascade

Structure	Type	Key	Value	Purpose
`_order_index`	dict	`order_id`	`Order`	Every resting order, stops included
`_entry_index`	dict	`order_id`	`_HeapEntry`	The live entry, for O(1) lazy deletion
`_orders_by_gateway`	dict	`gateway_id`	`dict[order_id, None]` (ordered set)	One gateway's resting orders, deterministic order
`_bid_qty`	dict	`price_ticks`	`int`	Visible bid quantity per level
`_ask_qty`	dict	`price_ticks`	`int`	Visible ask quantity per level
`_gtc_count`	int	—	—	Resting GTC orders, for `book_stats()`
`trace`	list or `None`	—	tuple	Records shared with the engine, published after each dispatch
`recent_trades`	deque	—	`Trade`	Last 20 trades (snapshot publishes 5)

6.33 Complexity Summary

Operation	Complexity	Notes
Rest an order	O(log n)	`heappush` + three index inserts + one level update
Find the best order	O(1) amortised	`_peek` pops each dead entry once
One fill	O(1) amortised	`_apply_fill`; a full fill adds a purge
Cancel	O(1) amortised	dict work + `valid = False`; occasional O(n) `_compact_heaps`, amortised O(1)
Amend, priority kept	O(1)	level index only
Amend, priority lost	O(log n)	retire old entry + push new
Iceberg refill	O(log n)	`_reinsert_iceberg` push
FOK pre-check	O(n log n)	sorted copy of the opposite heap
Stop check, nothing fires	O(1)	compare against `heap[0]`
Stop check, k fire	O(k log s)	k pops from a stop heap of size s; each fired stop is then processed as an order
Trailing-stop check	O(t)	walks every trailing stop
Orders of one gateway	O(k)	`_orders_by_gateway`
Best bid / ask price	O(P)	`max`/`min` over the level index
`depth_snapshot()`	O(P)	level index only
`snapshot()`	O(n)	walks every heap entry, dead ones included

n = heap entries on a side, P = distinct price levels (usually P n), k = one gateway's resting orders or the number of fired stops, s = armed stops, t = trailing stops.

Glossary →

CHAPTER 7

Order Flow Through the Engine: A NEW LIMIT Order, Step by Step

Learning objectives

After reading this page you will understand:

- The exact sequence of sockets and processes a `NEW` order crosses between a trader's keystroke and a resting entry in the book
- Every check the engine runs before an order can rest or trade, in the order it runs them, with the code that runs them
- How the engine's `ACK` finds its way back to the *one* gateway that sent the order — there is no per-connection routing table, just a topic string
- What actually happens to the order inside `OrderBook.process()` — resting, partial fill, full fill — with a worked numeric example
- Exactly when a `TIF=DAY` order leaves the book: on a fill, on an explicit cancel, on the scheduled session close — and, separately, what happens (and does **not** happen) to it across a process restart
- Where to look first when something about an order's fate is unclear

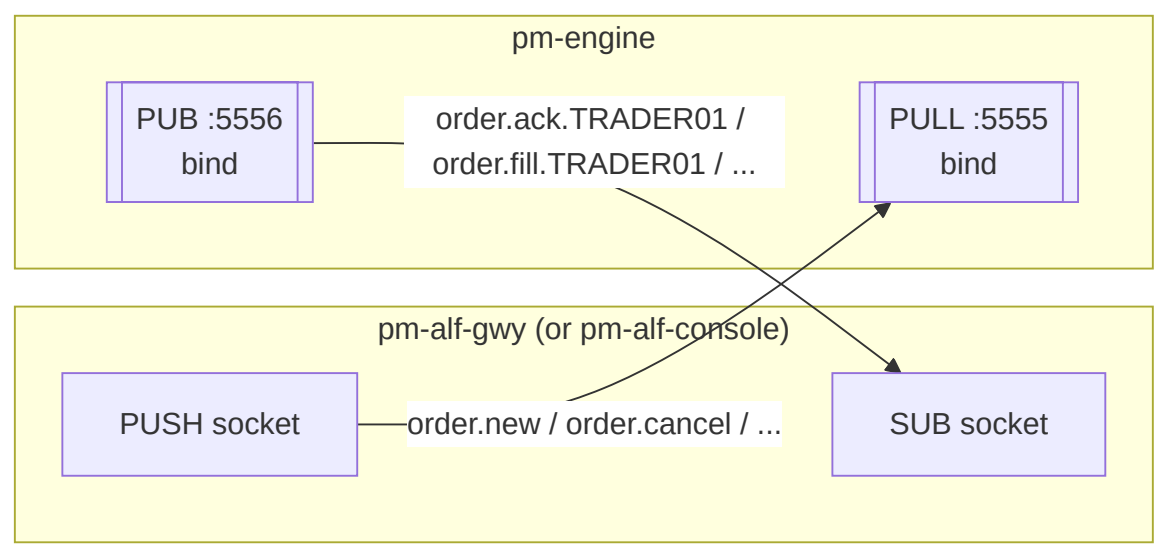
What is covered

This page walks one concrete case end to end: gateway `TRADER01` submits `NEW BUY 100 AAPL LIMIT @150.00` while the session is `CONTINUOUS`. Every code snippet below is taken from the real files (line numbers omitted — they drift; the surrounding context does not). Nothing here is simplified into a lie; where a snippet is trimmed for length, the omission is marked and the full function is named so you can go read it.

The goal is to make it boring. An exchange matching engine sounds like it should be full of clever tricks. It mostly is not — it is a long, explicit checklist, run in a fixed order, each step small enough to hold in your head. The complexity is in the *number* of steps, not in any single one of them.

7.1 1. The moving parts

Two ZeroMQ sockets connect every gateway process to the one engine process:



- **PULL (`:5555`)** — every gateway process `connect()`s a PUSH socket to it and sends commands *in*. Many gateways, one engine, one-way traffic.
- **PUB (`:5556`)** — the engine broadcasts *everything* it does. Every gateway `connect()`s a SUB socket and filters, by topic prefix, for only the events addressed to the gateway ids it is currently serving.

```
# messaging/bus.py
def make_puller(addr: str) -> zmq.Socket[bytes]:
    """PULL socket — engine receives orders."""
    sock = get_context().socket(zmq.PULL)
    sock.bind(addr)
    return sock

def make_publisher(addr: str) -> "CausalPublisher":
    """PUB socket — broadcasts events, sequenced and causally stamped."""
    sock = get_context().socket(zmq.PUB)
    sock.bind(addr)
    return CausalPublisher(SequencedPublisher(sock))
```

A message is two frames of **content** — a topic string and a JSON payload — followed by **envelope** frames that describe the message rather than carry it:

Frame	PUB (engine → subscribers)	PUSH (gateway → engine)
0	topic	topic
1	JSON payload	JSON payload
2	per-topic sequence	causal envelope
3	causal envelope	—

The envelope is metadata, not content, which is why it rides in frames instead of in the JSON: the hot publish path never decodes and re-encodes a message to stamp it, and the same three fields do not have to be declared on each of the 121 message types in `spec/messages/`. Sections 3, 4 and 8 show it being written and read; if you only care about the order flow, the one thing to know is that `decode()` ignores it entirely.

```
# models/message.py
def encode(topic: str, payload: dict[str, Any]) -> list[bytes]:
    return [topic.encode(), _dumps(payload)]

def decode(frames: list[bytes]) -> tuple[str, dict[str, Any]]:
    topic = frames[0].decode()
    payload = _loads(frames[1])
    return topic, payload
```

Keep that shape in mind — `(topic, payload)` — it is the unit everything below is built from. The envelope never changes it; `decode` reads frames 0 and 1 and stops, so every consumer written before the envelope existed still works unmodified.

7.2 2. The scenario

`TRADER01`, connected through `pm-alf-gwy`, types:

```
NEW|SYM=AAPL|SIDE=BUY|QTY=100|PRICE=150.00|TIF=DAY|TYPE=LIMIT
```

Session state is `CONTINUOUS`. AAPL's book currently has no resting bids and one resting ask at 150.50 for 200 shares. This order does not cross — it will rest. Section 7 below re-runs the same order at a price that *does* cross, to show a partial fill too.

7.3 3. From keystroke to the engine's doorstep

`pm-alf-gwy` parses the ALF text line, builds an `Order`, and pushes it — there is no engine-visible difference between an order that arrived over ALF, BALF, REST, or from `pm-alf-console` directly; they all end up as the same `order.new` PULL message.

```
# alf_gwy/gateway.py — _handle_new_single (trimmed)
order = Order.create(
    symbol=symbol,
    side=side,
    order_type=order_type,
    quantity=quantity,
    gateway_id=self._require_gw(session),   # ← the AUTHENTICATED session's id
    tif=tif,
    price=self._ticks(price, symbol, "PRICE") if price is not None else None,
    ...
)
self._send_to_engine(make_order_new_unchecked_msg(order.to_dict()))
```

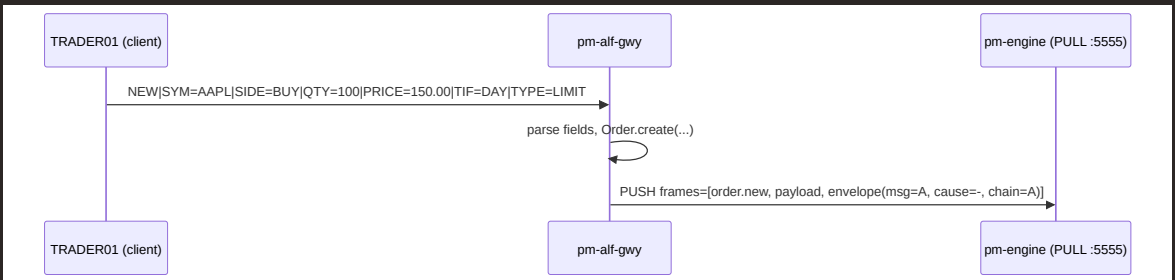
Two things worth pausing on:

- **`gateway_id` comes from the session, not from the wire fields.** A client cannot claim to be a different gateway than the one it authenticated as — `_require_gw(session)` raises if the session never completed `HELLO`. This is the fact the ACK routing in Section 6 depends on.
- **Price is already converted to ticks** (`self._ticks(...)`) before the order ever leaves the gateway process. The engine never does float arithmetic on a price — see `docs-design/EduMatcher-Requirements.md` section 15.2 if you want the full story on ticks vs. display money.

`_send_to_engine` is a thin wrapper around the PUSH socket:

```
# alf_gwy/gateway.py — _send_to_engine (trimmed)
def _send_to_engine(self, frames: list[bytes], ...) -> None:
    self._push.send_multipart(frames)
```

This send is where the causal chain begins. `make_pusher` returns a `CausalPusher`, which appends an envelope frame carrying a fresh `msg_id` (a ULID), no `causation_id` — nothing on the bus caused this, the trader typed it — and a `correlation_id` equal to its own `msg_id`. Everything the engine publishes in response will cite that id and carry that chain, which is what makes "show me everything this submission caused" a lookup rather than a reconstruction. The gateway code above does not mention any of it: wrapping the socket is what stops thirteen client processes each having to remember. See `models/envelope.py` and Architecture — Causal envelope.



7.4 4. The engine's front door

The engine's whole life is one loop: poll the PULL socket, decode, dispatch, run maintenance, repeat.

```
# engine/main.py — run() (trimmed)
while self._running:
    socks = dict(poller.poll(timeout=200))          # 200 ms tick
    if self.pull_sock in socks:
        frames = self.pull_sock.recv_multipart()
        topic, payload = decode(frames)
        cause = decode_push_envelope(frames)        # the gateway's msg_id

        self.pub_sock.set_cause(cause)
        try:
            self._dispatch_pull_message(topic, payload)
        finally:
            self.pub_sock.clear_cause()
    self._run_maintenance()
```

Those three extra lines are the entire causality mechanism on the engine side. While a cause is set, **everything** published — the ACK, the fills, the trade print, a cascade of OCO cancels — is attributed to this message and inherits its chain, without a single publish site knowing. Three things about that are worth understanding rather than skimming:

- **`decode()` still reads only frames 0 and 1.** The envelope rides behind them, so nothing downstream of the payload changed.
- **The `finally` is load-bearing.** If a handler raises, the cause must be cleared anyway — otherwise `_run_maintenance()` below, which has no external cause at all, would publish scheduler ticks and circuit-breaker trips attributed to an order that had already failed.
- **This is safe only because the loop is single-threaded.** Exactly one inbound message is in flight, so there is never more than one cause in scope. A multi-threaded publisher would need a context variable, and `CausalPublisher`'s docstring says so.

A message published with **no** cause in scope is not missing information: it declares `causation_id = null`, which is the engine positively stating that it decided this by itself.

`_dispatch_pull_message` is a plain `if/elif` chain from topic string to handler method — no registry, no magic, just a long ladder:

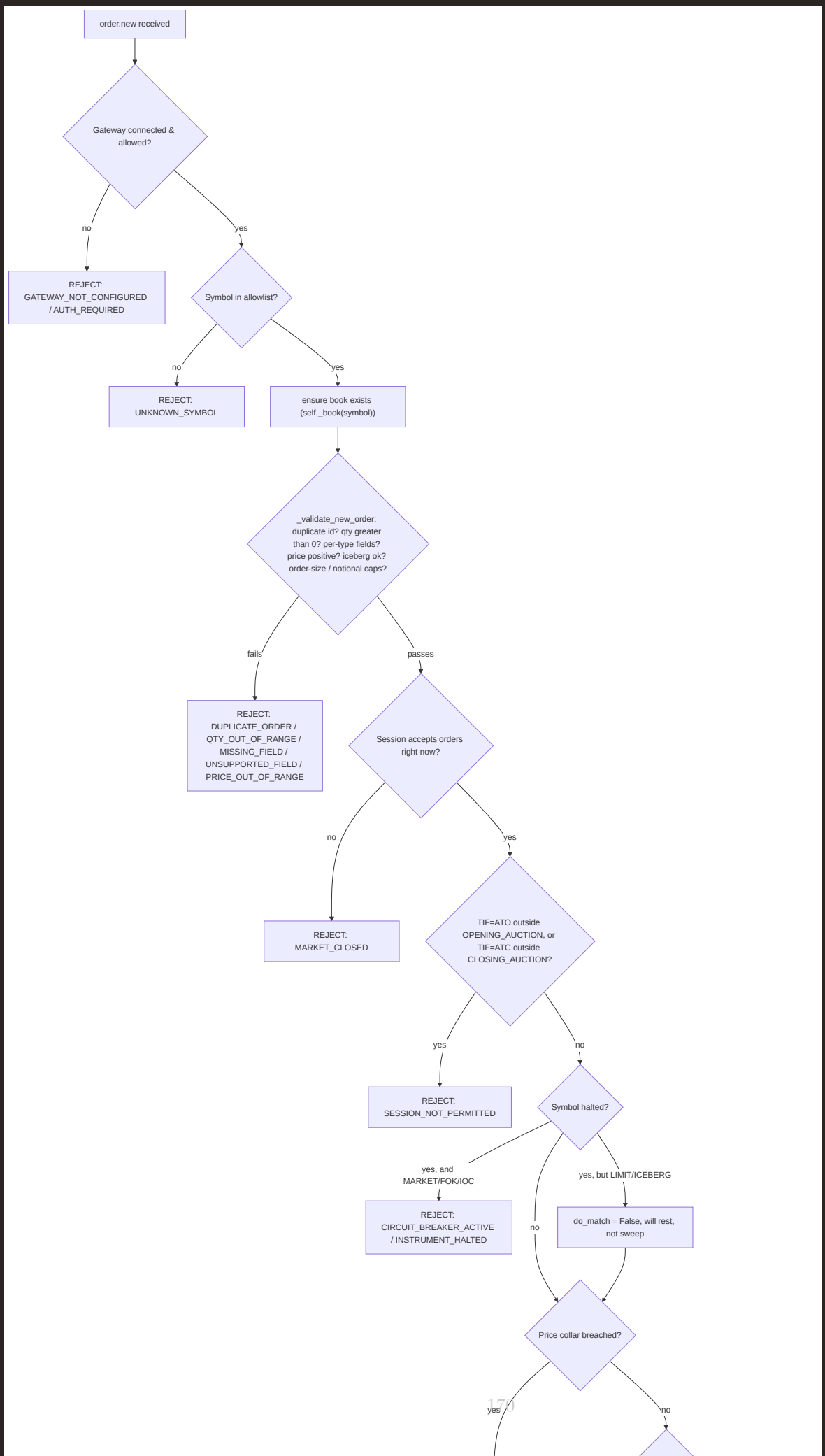
```
# engine/main.py — _dispatch_pull_message (trimmed)
try:
    if topic == TOPIC_ORDER_NEW:
        self._handle_new_order(payload)
    elif topic == TOPIC_ORDER_CANCEL:
        self._handle_cancel(payload)
    elif topic == TOPIC_ORDER_AMEND:
        self._handle_amend(payload)
    ...
    else:
        self._unknown_topic_count += 1
        log.warning("No dispatch handler for topic %s ...", topic)
except Exception as exc:
    self._error_count += 1
    log.error("Error processing %s (#%d): %s", topic, self._error_count, exc)
    self._reject_after_error(topic, payload, fills_before)
```

The `try/except` around the whole ladder is deliberate: one malformed or unexpected payload must never take the process down, and it must never leave the submitting gateway simply waiting forever for an ACK that will never come — `_reject_after_error` sends a `REJECTED` with `code=INTERNAL_ERROR` so the caller's wait always terminates in *something*.

Our message matches `TOPIC_ORDER_NEW`, so `_handle_new_order(payload)` runs next. Everything from here to "resting or trading" happens inside that one method.

7.5 5. The gauntlet: every check, in order

`_handle_new_order` is a sequence of guard clauses. Each one either rejects (sends a NACK and returns) or falls through to the next. Below is the exact order, each with the real check.


order.new received
Gateway connected & allowed?
no
yes
REJECT: GATEWAY_NOT_CONFIGURED / AUTH_REQUIRED
Symbol in allowlist?
no
yes
REJECT: UNKNOWN_SYMBOL
ensure book exists (self._book(symbol))
_validate_new_order: duplicate id? qty greater than 0? per-type fields? price positive? iceberg ok? order-size / notional caps?
fails
passes
REJECT: DUPLICATE_ORDER / QTY_OUT_OF_RANGE / MISSING_FIELD / UNSUPPORTED_FIELD / PRICE_OUT_OF_RANGE
Session accepts orders right now?
no
yes
REJECT: MARKET_CLOSED
TIF=ATO outside OPENING_AUCTION, or TIF=ATC outside CLOSING_AUCTION?
yes
no
REJECT: SESSION_NOT_PERMITTED
Symbol halted?
yes, and MARKET/FOK/IOC
yes, but LIMIT/ICEBERG
no
REJECT: CIRCUIT_BREAKER_ACTIVE / INSTRUMENT_HALTED
do_match = False, will rest, not sweep
Price collar breached?
yes
no

The code, one gate at a time:

Gate 1 — gateway connected and allowed.

```
ok, reason = self._gateway_status(order.gateway_id)
if not ok:
    self._reject(gateway_id=order.gateway_id, order_id=order.id,
                 code=self._gateway_reject_code(reason), reason=reason, ...)
    return
```

Gate 2 — symbol allowlist.

```
if self._allowed_symbols and order.symbol not in self._allowed_symbols:
    self._reject(..., code="UNKNOWN_SYMBOL",
                 reason=f"Symbol not configured: {order.symbol}", ...)
    return
```

Gate 3 — boundary validation, all in one place (`_validate_new_order`), run *before* any ACK so a malformed order never reaches the book:

```
# engine/main.py — _validate_new_order (trimmed)
if order.id in self._order_symbol:
    return "DUPLICATE_ORDER", f"Duplicate order id {order.id}"
if order.quantity <= 0 or order.remaining_qty <= 0:
    return "QTY_OUT_OF_RANGE", "Quantity must be positive"
violation = check_type_fields(order.order_type, present_fields)  # models/order_fields
if violation is not None:
    return violation.reject_code, violation.message  # MISSING_FIELD / UNSUPPORTED_FIELD
if order.price_ticks is not None and order.price_ticks <= 0:
    return ("PRICE_OUT_OF_RANGE",
            f"{order.order_type.value} order requires a positive price")
# ... ICEBERG visible_qty checks ...
# ... order-size / notional cap check via validate_order_limits() ...
return None
```

The per-type rule (`models/order_fields.py`) is the one REST, ALF and BALF already applied before sending: which of `price`, `stop_price`, `visible_qty` and `trail_offset` each order type needs, may carry, or must not carry. The engine applies it again as the backstop for every transport, to OCO and combo legs too, so a mistake gets the same `reject_code` however it arrives.

Our order is `LIMIT`, `quantity=100`, `price=15000` ticks — every check here passes.

Gate 4 — session gating.

```
if self._sessions_enabled and not accepts_orders(self._session_state):
    self._reject(..., code="MARKET_CLOSED", reason="Market is closed", ...)
    return
```

Session is `CONTINUOUS`, which `accepts_orders()` allows.

Gate 5 — ATO/ATC window. `TIF=DAY` skips both of these gates outright (they only fire for `TIF=ATO` outside `OPENING_AUCTION` and `TIF=ATC` outside `CLOSING_AUCTION`).

Gate 6 — halt check. AAPL is not halted, so this falls straight through.

Gate 7 — price collar. If a collar is configured for AAPL, `validate_collar` compares 150.00 against the reference price and the configured band. Assume it passes.

Gate 8 — no-match-phase rejection. Only bites `MARKET`/`FOK`/`IOC` orders when `do_match` is `False` (auction phase or halted symbol forcing LIMIT-only resting). Our order is `LIMIT`, so this is a no-op either way.

Every gate passed. The order is now **accepted**.

7.6 6. The ACK — sent before the book ever sees the order

This is the detail most people expect to be complicated and is not: the ACK is built and published *before* `book.process()` runs at all (for every order type but `FOK`, see the caveat below).

```
# engine/main.py — _handle_new_order (trimmed)
_gw = order.gateway_id                          # "TRADER01"
ack_topic = self._topic_cache.get(_gw)
if ack_topic is None:
    self._topic_cache[_gw] = topic_order_ack(_gw).encode()   # b"order.ack.TRADER01"
    ack_topic = self._topic_cache[_gw]

self.pub_sock.send_multipart([
    ack_topic,
    dumps({
        "order_id": order.id,
        "accepted": True,
        "reason": "",
        "symbol": order.symbol,
        "side": _side_v,
        "order_type": _ot_v,
        "tif": _tif_v,
        "qty": order.quantity,
        "price": _price_v,
        "client_tag": order.client_tag,
    }),
])
```

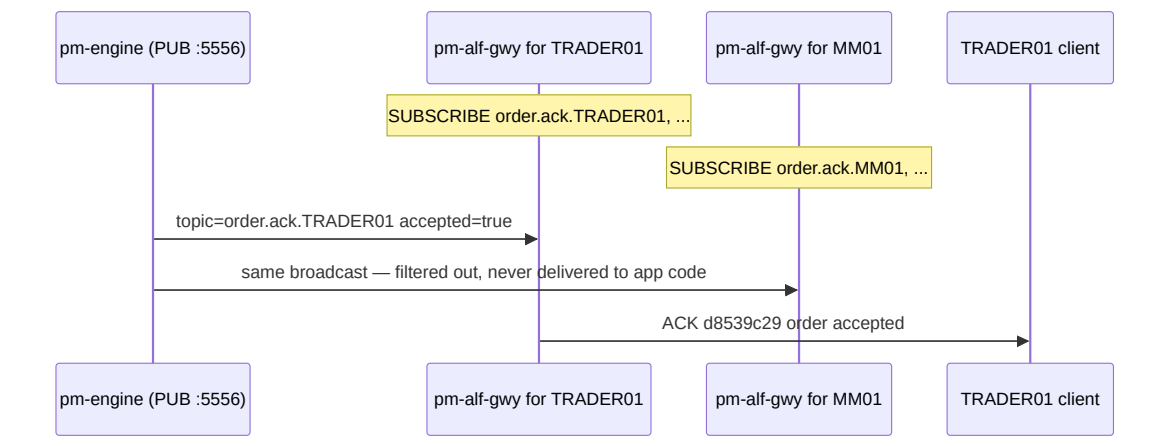
How does the engine know which gateway to answer? It doesn't look anything up — `order.gateway_id` is already `"TRADER01"`, carried in the payload since Section 3 (`self._require_gw(session)`), and it becomes half of the outbound *topic string*: `order.ack.TRADER01`. There is no per-connection socket or address to remember on the engine side at all.

And which submission is it answering? Note what the snippet above does *not* contain: anything about correlation. The `send_multipart` call goes through the `CausalPublisher` wrapper, which stamps `causation_id` = the submitting message's `msg_id` from the cause Section 4 set. So the ACK answers a specific *message*, not merely a gateway — which matters the moment a trader has two orders in flight and both acks route to the same topic. Before the envelope this had to be inferred by matching `order_id` between the submission and the ack; now the engine states it.

Delivery is ZeroMQ's job, and it works by **prefix filtering on the subscriber side**, not by the publisher picking a destination:

```
# alf_gwy/gateway.py — _gateway_topics
def _gateway_topics(self, gateway_id: str) -> tuple[str, ...]:
    return (
        topic_gateway_auth(gateway_id),
        topic_order_ack(gateway_id),      # "order.ack.TRADER01"
        topic_order_fill(gateway_id),
        ...
    )
```

When `TRADER01` authenticates, `pm-alf-gwy` calls `SUBSCRIBE` on exactly those topic strings. The engine's PUB socket broadcasts to *every* gateway process indiscriminately; each one's SUB socket silently drops anything that does not match one of its subscribed prefixes. `MM01`'s gateway process never even receives `order.ack.TRADER01` — ZeroMQ filters it at the network layer before it reaches Python.



`pm-alf-gwy` then relays the ACK into the ALF text protocol (`_dispatch_response` → `ACK|ORDER_ID=...|ACCEPTED=TRUE`), and `TRADER01`'s terminal prints it.

One caveat: this ACK means *"the engine accepted the order for processing"*, not *"this is its final state"*. The exception is a `FOK`, which the book can still reject for want of liquidity: its ACK is sent *after* `book.process()` (`_book_may_reject`), so an unfillable `FOK` gets a single `accepted=False` ACK with `INSUFFICIENT_LIQUIDITY` and never an acceptance first. For our resting `LIMIT` order, this ACK is the only acknowledgement it gets; its fate from here is reported via `order.fill` / `order.cancelled` / `order.expired`.

7.7 7. Handing the order to the book

```
# engine/main.py — _handle_new_order (trimmed)
do_match = is_matching_enabled(self._session_state)   # True in CONTINUOUS
trades, events = book.process(order, match=do_match, now=now)
```

`OrderBook.process` dispatches on `order_type`:

```
# engine/order_book.py — process (trimmed)
elif order.order_type == OrderType.LIMIT:
    self._match_limit(order, trades, events, now)
```

```
# engine/order_book.py — _match_limit
def _match_limit(self, order, trades, events, now):
    opposite = self._asks if order.side == Side.BUY else self._bids
    self._sweep(order, opposite, price_limit=order.price_ticks,
                trades=trades, events=events, now=now)
    if order.remaining_qty > 0 and order.status not in _DEAD_STATUSES:
        self._rest(order)
```

`_sweep` is one `while` loop: peek the best opposite order, stop if its price is worse than our limit, otherwise fill against it and keep going until either side runs out.

```
# engine/order_book.py — _sweep (trimmed)
while aggressor.remaining_qty > 0 and opposite_heap:
    best = self._peek(opposite_heap)
    if best is None:
        break
    if price_limit is not None:
        if side == Side.BUY and best.price_ticks > price_limit:
            break                       # nothing left is cheap enough — stop
        if side == Side.SELL and best.price_ticks < price_limit:
            break
    fill_qty = min(aggressor.remaining_qty, best.remaining_qty)
    self._apply_fill(aggressor, best, fill_qty, best.price_ticks, trades, events, now)
```

That's the entire matching algorithm for a LIMIT order: **peek, compare price, fill the smaller of the two remaining quantities, repeat.** No special cases beyond price-time priority (best price first, then arrival order at a tied price — see

the heap key in `_rest` below) and self-match prevention (skipped here for brevity — see The First Match for the full sweep including SMP).

7.7.1 7a. Outcome 1 — the order rests (our scenario)

AAPL's best ask is 150.50; our bid is 150.00. `150.50 > 150.00`, so the very first price check in `_sweep` breaks the loop immediately — nothing trades. Back in `_match_limit`, `order.remaining_qty` is still 100, so it rests:

```
# engine/order_book.py — _rest (trimmed)
def _rest(self, order: Order) -> None:
    price = order.price_ticks
    order.arrival_seq = self._next_seq()          # time priority (H1)
    if order.side == Side.BUY:
        key = (-price, order.arrival_seq)         # highest price first
        heap = self._bids
        self._bid_qty[price] = self._bid_qty.get(price, 0) + order.remaining_qty
    else:
        key = (price, order.arrival_seq)          # lowest price first
        heap = self._asks
        ...
    heapq.heappush(heap, _HeapEntry(key=key, order=order))
    self._order_index[order.id] = order
    self._entry_index[order.id] = entry
```

That's the whole "resting order" concept: a heap entry keyed `(-price, arrival_seq)` for bids (so `heapq`, a min-heap, naturally pops the *highest* price first) or `(price, arrival_seq)` for asks (lowest price first), plus two dict lookups (`_order_index`, `_entry_index`) so a later `CANCEL`/`AMEND` can find it in O(1) instead of scanning a heap.

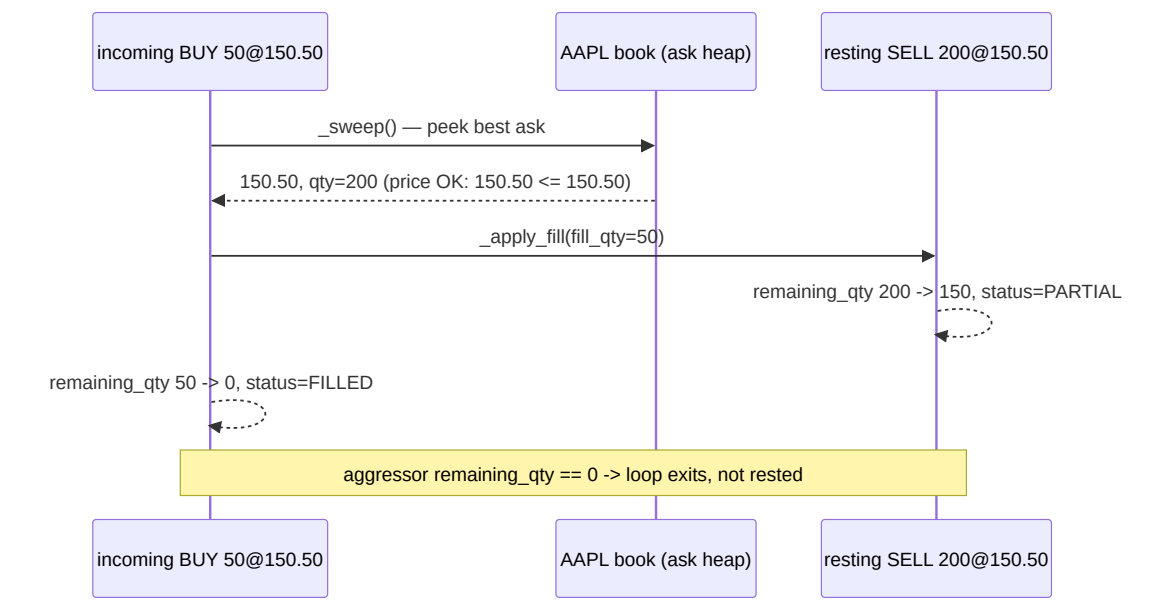
`book.process` returns `trades=[]`, `events=[]` — nothing to publish. The book is now dirty (marked via `self._mark_dirty(order.symbol`) and will appear in the next throttled `book.AAPL` snapshot.

7.7.2 7b. Outcome 2 — a crossing order (worked example)

Suppose instead `TRADER01` had bid **150.50 for 50 shares** — at or above the resting ask. `_sweep`'s price check now passes (`150.50` is not `> 150.50`), so it fills:

```
fill_qty = min(50, 200) = 50
_apply_fill(aggressor=bid, resting=ask@150.50, fill_qty=50, price=150.50, ...)
```

`_apply_fill` (not reproduced in full — see `order_book.py`) does three things: creates a `Trade`, decrements both orders' `remaining_qty`, and sets each order's `status` (`FILLED` if `remaining_qty` hits zero, `PARTIAL` otherwise). The aggressor's `remaining_qty` is now 0 — the loop exits because `aggressor.remaining_qty > 0` is false — and since it is not `> 0`, `_match_limit` does **not** rest it. The resting ask had 200 shares and is now down to 150, still resting with `status=PARTIAL`.



`trades` now has one `Trade`, `events` has both `Order` objects (the filled aggressor and the partially-filled resting order) — Section 8 covers what happens to them next.

7.8 8. Publishing what happened

Back in `_handle_new_order`, `trades` and `events` (every order whose status just changed) go to `_publish_dispatch`. Every path that matches calls the same method – a quote, a combo child, an OCO leg, an amend that crosses, an auction uncross – so what a participant is told cannot depend on how the order that hit them arrived. The important subtlety is deduplication: a single aggressive order that sweeps several price levels appears **once per level** in `events`, each occurrence carrying the same order object with its *final*, cumulative state — so a naive loop would publish the same fill several times with the wrong quantity split. Each order is handled once:

```
# engine/main.py — _publish_dispatch (trimmed)
executed = self._executions(trades)   # order_id -> qty, VWAP, trade ids, flag
                                      # in THIS dispatch only
seen: set[str] = set()
for evt in events:
    if evt.id in seen:
        continue
    seen.add(evt.id)
    ex = executed.get(evt.id)
    if ex is not None:                # it traded here, whatever its status now
        send(self._fill_frames(evt, ex))
    if evt.status is OrderStatus.CANCELLED:
        send(...)                     # order.cancelled with its cancel_reason
    elif evt.status is OrderStatus.REJECTED:
        self._reject(...)             # e.g. a FOK short of liquidity
    ...                               # combo / OCO / quote follow-ups
for trade in trades:
    self._publish_trade(trade)        # trade.executed + both sides' positions
```

`fill_qty` is "matched in this dispatch," not `evt.quantity - evt.remaining_qty`. The obvious formula is the order's *cumulative* fill since it first rested — correct only the first time an order fills, and silently wrong (reporting the running total again) the second time a resting order is hit. `order.fill.fill_qty` is specified as quantity matched in *this* event, not cumulatively (`spec/messages/order.yaml`), so `_executions(trades)` builds the per-order increment directly from this dispatch's own `trades` list instead — the sum of every trade in `trades` that named this order as either side. This needs no "before" state and is correct for a first fill, a second hit on a still-resting order, and everything in between.

`_publish_trade` is the single fan-out point for a trade: it broadcasts the public `trade.executed` tape event, updates **both** counterparties' position ledgers (`_update_position`), feeds the drop-copy relay if one is attached, and checks whether this print

should trip a circuit breaker. This is the *only* place any of those four things happen, regardless of whether the trade came from a plain NEW order, a quote leg, a combo child, an OCO leg, or an auction uncross — one path, four side effects, always in that order.

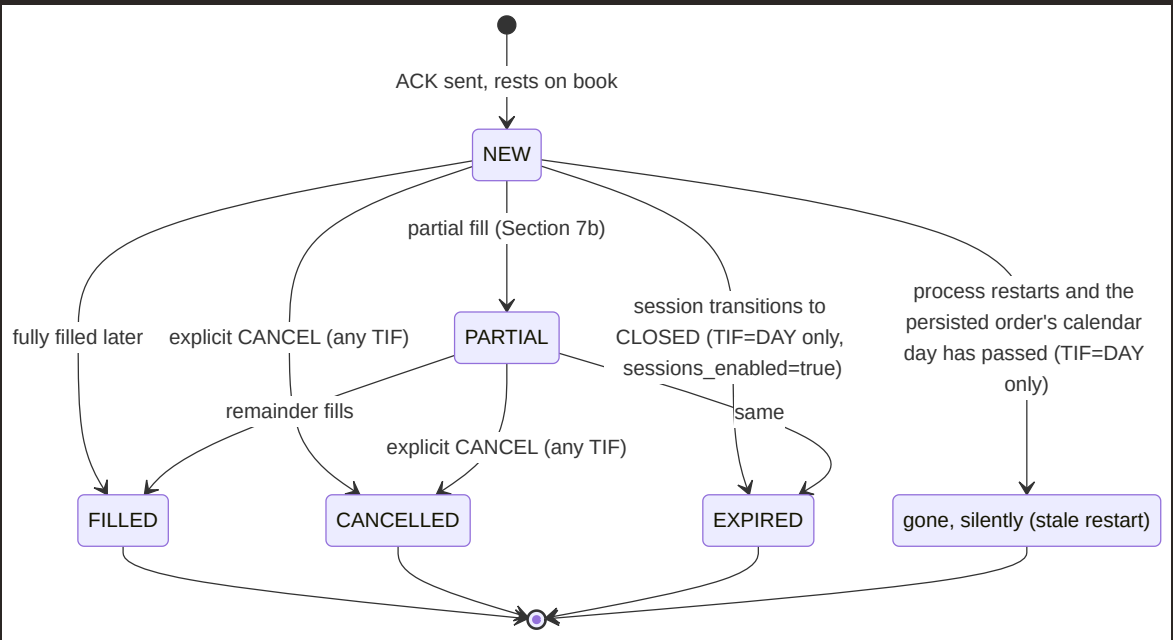
Every one of those messages — the ACK from Section 6, each `order.fill`, the `trade.executed` print, the drop-copy relay — carries the same `correlation_id`, because all of them are published inside the dispatch window Section 4 opened. One aggressive order that sweeps four price levels produces a dozen messages across five topics and two gateways, and they are one `WHERE correlation_id = ?` away from each other. That is the payoff of attributing at the socket rather than at the publish site: the fan-out did not have to thread anything through to get it.

Note also what this does *not* claim. The counterparty's `order.fill` is caused by **our** submission, and says so — the resting order's own submission, minutes earlier, is a different chain. Causality here means "this message exists because that one was processed", not "these belong to the same trader".

If nothing filled (our resting-order scenario from 7a), `events` and `trades` are both empty — the loop above simply does nothing, and the resting order's only trace on the wire so far is the ACK from Section 6.

7.9 9. TIF and the lifetime of a resting order

Our order rested with `TIF=DAY`. What removes it, and when?



There are exactly four ways a `TIF=DAY` order stops resting. Two are the same for every TIF; two are DAY-specific.

1 & 2 — fill and explicit cancel. Nothing DAY-specific here: a fill happens the moment a crossing order sweeps it (Section 7b); a cancel happens the moment the owning gateway sends `CANCEL` (`Engine._handle_cancel`). Both apply identically to `GTC`.

3 — the scheduled end of the trading day, *if and only if* `sessions_enabled` is `true` and `pm-scheduler` actually drives a transition to `CLOSED`:

```
# engine/main.py — _handle_session_transition (trimmed)
self._session_state = to_state
if to_state == SessionState.CLOSED:
    self._expire_tif(TIF.DAY)     # <- the only caller of this, for TIF.DAY
```

```
# engine/main.py — _expire_tif
def _expire_tif(self, tif: TIF) -> None:
    for book in self.books.values():
        for order in book.resting_orders():
            if order.tif == tif:
                cancelled = book.cancel_order(order.id)
                if cancelled:
                    cancelled.status = OrderStatus.EXPIRED
                    self.pub_sock.send_multipart(make_expired_msg(
                        cancelled.gateway_id, cancelled.id,
                        order=cancelled.to_dict(),
                    ))
```

This is the *only* code path that ever publishes `order.expired` for a `TIF=DAY` order. If `sessions_enabled` is `false` (many training/demo configs run this way), **this path never runs at all** — nothing in the engine ever expires a DAY order on a clock. It rests exactly as long as a `GTC` order would, until something else removes it.

4 — a stale restart, which is a completely different mechanism and does *not* publish anything.

7.9.1 What happens at shutdown — and it is *not* removal

```
# engine/main.py — _shutdown (trimmed)
def _shutdown(self) -> None:
    all_resting = self._resting_gtc_orders()      # TIF in (GTC, DAY)
    save_gtc_orders(all_resting, GTC_ORDERS_FILE)
    save_book_stats(self.books, BOOK_STATS_FILE)
    ...
```

```
# engine/persistence.py — save_gtc_orders
gtc = [
    o.to_dict() for o in orders
    if o.tif in (TIF.GTC, TIF.DAY)
    and o.status in (OrderStatus.NEW, OrderStatus.PARTIAL)
]
_atomic_write_text(path, json.dumps(gtc, indent=2))
```

A `TIF=DAY` order is persisted to `data/gtc_orders.json` on shutdown, exactly like a `GTC` order. Nothing is expired, nothing is discarded — a clean shutdown (SIGINT/SIGTERM) and a crash both leave the file in this state (a crash loses at most `_PERSIST_INTERVAL_SEC` = 5 seconds of changes, via the periodic `_flush_persistence` checkpoint that runs the same save on every maintenance tick — see `_run_maintenance`). **A process exit is not a day boundary.**

7.9.2 What happens at the next startup — this is where staleness is decided

```
# engine/main.py — _restore_gtc (trimmed)
today = datetime.now().date()
for order in orders:                                  # loaded from the JSON file
    if order.tif == TIF.DAY:
        order_day = datetime.fromtimestamp(order.ts_ns / 1e9).date()
        if order_day < today:
            log.info(f"Discarding stale TIF=DAY order {order.id[:8]} ...")
            continue                                   # <- never restored, never expired
    order.status = OrderStatus.NEW
    book.process(order, match=False)                  # rest it, no sweep
```

The check is a plain calendar-date comparison against the order's original `timestamp`, evaluated **once, at process startup** — not continuously, not at midnight, not tied to the scheduler in any way. Two consequences follow directly from that:

- If the engine is restarted several times *within* the same calendar day, a `TIF=DAY` order survives every restart unchanged — the date comparison keeps passing.
- If the engine simply never restarts, a `TIF=DAY` order with `sessions_enabled=false` never expires at all, ever — there is no wall-clock timer for it, only the two checks above (session-close, or a restart after the date has rolled over).

And critically: **a stale order discarded here never generates an `order.expired` message.** By the time this code runs, the PUB socket may not even have a single subscriber yet — the order is simply absent from the book that gets rebuilt, and

absent from the file the next time it is re-saved. A gateway reconnecting after a multi-day outage learns about the loss only by *absence*: it is not in the `ORDERS` snapshot it requests on reconnect, not in `QLEGS` (Section 10 below), and no lifecycle event for it will ever arrive.

7.9.3 Summary table

Trigger	Applies to	Publishes `order.expired`?	Where in the code
Fill (full or partial)	any TIF	no — `order.fill` instead	`OrderBook._apply_fill`
Explicit `CANCEL`	any TIF	no — `order.cancelled` instead	`Engine._handle_cancel`
Session → `CLOSED`	`TIF=DAY` (and `ATO`/`ATC` at their own phase boundaries)	**yes**	`Engine._expire_tif`, called from `_handle_session_transition`
Process shutdown (clean or crash)	none — it is persisted, not removed	no	`Engine._shutdown` / `_flush_persistence`
Process restart, order's date has passed	`TIF=DAY` only	no — silently dropped	`Engine._restore_gtc`

7.10 10. Debugging cheat-sheet

When an order did something you did not expect, these are the places to look, roughly in the order you should look at them:

Question	Where to look
Was the order rejected, and why?	The `REJECT_CODE`/`reason` on the `ACK` — every reject in Section 5 goes through `Engine._reject`, which always sets both
Did the engine even see it?	Engine log at `INFO`: `NEW {id} {symbol} {side} ...` is logged right before `book.process` in `_handle_new_order`
Did it fill, partially or fully?	`order.fill.{gateway_id}` on the wire; `trade.executed` for the public tape; engine log `TRADE {id} {symbol} qty=... @...`
Is it still resting?	`ORDERS` (ALF) / `order.orders_request` — walks `book.resting_orders()` directly, so it can never disagree with the book
Did it expire, and when?	`order.expired.{gateway_id}`; only ever sent from `_expire_tif`, only for a real session transition to `CLOSED`
Did I lose it across a restart?	Check the engine's startup log for `Discarding stale TIF=DAY order ...` — if you see it, that order is gone and nothing else will tell you
Is a handler silently swallowing exceptions?	It cannot — every branch of `_dispatch_pull_message` is wrapped, and any exception increments `self._error_count` and logs at `ERROR` with the topic name
Fine-grained counters for a whole class of rejects	`self._debug_counts` (`new_order_reject_gateway`, `_reject_symbol`, `_reject_validation`, `_reject_session`, `_reject_halt`, `_reject_collar`, `_reject_no_match_phase`, …) — logged as one summary line every 5 seconds at `DEBUG` via `_flush_debug_summary`
What did this one submission cause?	Grep the audit log for its `chain=` id — every descendant carries it. This replaces walking `order_id` → `trade_ids` → counterparty `order_id` by hand

Question	Where to look
Why was this message published at all?	Its `cause=` id names the message that caused it. Grep for that `msg=` to find it. A *missing* `cause=` is an answer too: the engine decided it alone
Did the bus drop anything?	`seq=` in the audit line is dense **per topic**, so a jump means messages were lost on that topic. PUB/SUB drops silently once a subscriber falls behind; this is the only thing that reveals it

7.10.1 Following a chain

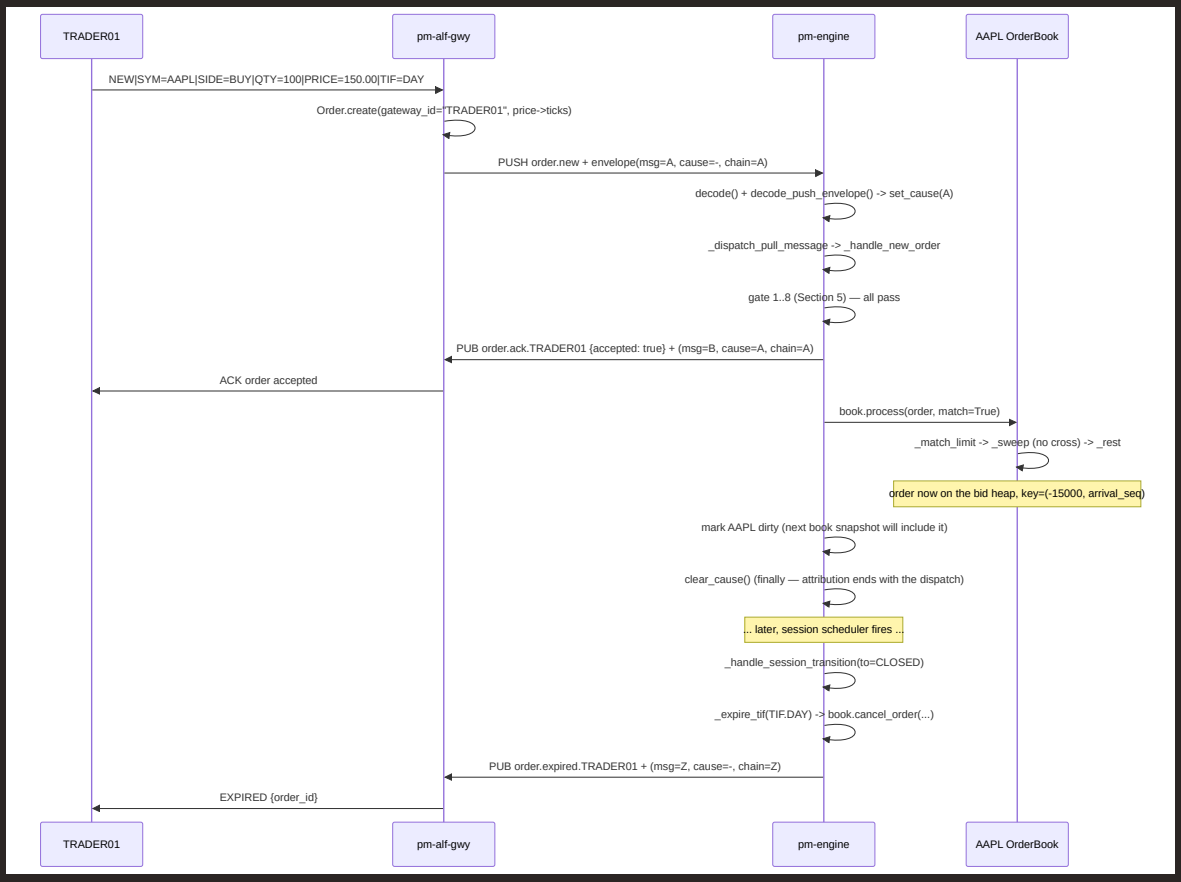
The envelope turns the awkward part of a post-mortem into two greps:

```
# 1. find the submission
grep 'order.new' data/audit.log | grep 'AAPL' | tail -1
# ... [order.new] [seq=8812 msg=01ARZ3NDEKTSV4RRFFQ69G5FAV chain=01ARZ3ND...] {...}

# 2. everything it caused, in order — acks, fills, trade prints, cascades,
#    across every topic and both counterparties
grep 'chain=01ARZ3NDEKTSV4RRFFQ69G5FAV' data/audit.log
```

The lines come back in audit-receipt order. For true publication order, sort by the `msg=` id: ULIDs are minted in publish order, so lexicographic order is the order the engine did things — across all topics, not just one.

7.11 11. The whole thing, end to end



Note the last message. The expiry carries **no `cause`**: hours after the submission, a session transition removed the order, and nothing the trader sent caused it. A new chain starts there, and that is the honest record — the alternative, attributing the expiry to the original submission, would make an automatic venue action look like a consequence of something the trader did.

Every arrow in that diagram is a function call or a ZMQ message you can grep for by name. The messages are two frames of content — topic and payload — plus the envelope frames behind them (`[topic, payload, seq, envelope]` on PUB, `[topic, payload, envelope]` on PUSH). There is no step in between.

CHAPTER 8

Development Practice

Learning objectives

After reading this page you will understand:

- How to set up a local Python development environment for EduMatcher
- Which quality gates are expected before you commit or release
- How the code base is structured and where to start reading it
- Which helper scripts exist under `scripts/` and `tools/`, and when to use them
- How to run a minimal exchange while developing
- How to run deterministic verification and performance tests
- How the current release workflow is intended to work
- Which common developer mistakes are easy to avoid

8.1 What kind of project you are joining

EduMatcher is a **multi-process educational exchange**. It is not just a single matching function or a toy order-book exercise. The repository contains:

- a matching engine
- interactive gateways
- market-data and monitoring processes
- persistence and reporting components
- developer tooling, benchmarks, and deterministic verification tools

That matters for development style. Most changes should be thought about at three levels:

1. **Core matching logic** — order validation, book state, fills, session rules
2. **Process boundaries** — ZeroMQ topics, startup order, persistence, observers
3. **Documentation and operability** — can another developer understand, run, and verify the change?

If you are new to the code base, start by reading these pages in order:

1. How an Exchange Works
2. Running the Exchange
3. Architecture Overview
4. Verification

8.2 Development environment

8.2.1 Recommended Python version

`pyproject.toml` requires Python `^3.14`, and the type-checking and formatting configuration targets the same version — so **Python 3.14 is the development target**, not merely the safest one. An earlier `^3.11` floor was raised; code in the tree now uses 3.12+ syntax (PEP 701 nested f-strings, for one), so an older interpreter fails at import rather than at runtime.

8.2.2 Canonical setup

The project is **Poetry-first**. The most reliable setup path is:

```
# Clone the repo
git clone https://github.com/johan162/EduMatcher.git
cd EduMatcher

# Verify necessart environment
./scripts/verify_setup.sh

# Setup poetry && Python
poetry config virtualenvs.in-project true
poetry install --with dev,docs

# Verify that a full build (incl. documents) succeeds
./scripts/mkbld.sh --intro
```

Then activate the environment if you want shell-local tools:

```
source .venv/bin/activate
```

8.2.3 Basic toolchain expectations

You should have these available locally:

Tool	Why you need it
Python 3.14	Matches the repo's linting and typing targets
Poetry	Dependency and virtualenv management
Git	Branching, release, and tag workflow
`gh`	Used by the GitHub release script
MkDocs	Installed through Poetry docs dependencies

If you plan to use the containerised docs workflow, you also need **Podman** for `scripts/docs-contctl.sh`.

8.3 Repository map

When you are orienting yourself, this is the practical top-level map:

Path	What lives there	Typical reason to open it
`src/edumatcher/engine/`	Matching engine, config loading, persistence, risk logic	Core exchange behavior
`src/edumatcher/alf_gwy/`, `balf_gwy/`	Protocol gateways: ALF (text) and BALF (binary) order entry	Trader entry workflow
`src/edumatcher/alf_console/`	Interactive trader console and command parsing	Hands-on order entry, demos
`src/edumatcher/api_gateway/`	REST/WebSocket edge for the web GUIs	Browser-facing flows
`src/edumatcher/md_gateway/`, `dc_gateway/`, `ralf_gateway/`	Market-data (CALF), drop-copy and RALF relay gateways	Feed and post-trade consumers
`src/edumatcher/alf_client/`, `calf_client/`	In-process client libraries for ALF order entry and the CALF market-data feed	Writing a tool or agent that talks to a gateway (see below)
`src/edumatcher/commands/`	Admin/console command clients and tooling	Operator workflows and scripted control

Path	What lives there	Typical reason to open it
`src/edumatcher/messaging/`	Transport and message-bus helpers	Socket wiring and topic flow
`src/edumatcher/models/`	Shared message, order, and domain models	Data structures and message payloads
`src/edumatcher/clearing/`	P&L and trade-settlement logic	Post-trade reporting
`src/edumatcher/ai_trader/`	AI trader and swarm entry points	Agent-based flow and experiments
`src/edumatcher/viewer/`, `board/`, `ticker/`, `orders/`, `audit/`, `stats/`	Read-side processes and UIs	Operational visibility
`src/edumatcher/scheduler/`	Session transitions	Trading-day lifecycle
`src/edumatcher/setup_cmd.py`	`pm-setup` bootstrap command	Runtime setup for installed mode
`tests/`	Unit, integration, and performance tests	Regression protection
`tools/`	Verification utilities and launch helpers	Deterministic validation, scripted demos
`scripts/`	Build, release, docs, and maintenance helpers	Automation
`docs/`	User, architecture, concept, and developer documentation	Documentation updates
`docs-design/`	Design proposals and implementation plans	Architecture and feature design review
`release_checklist.md`	Canonical release checklist	Pre-release gate and release sequencing

8.3.1 Client libraries: `alf_client` and `calf_client`

These two packages are libraries for code *inside this repository* that needs to talk to a gateway. They are not commands and have no entry point, so an end user never runs them.

Package	Protocol	Contents	Used by
`alf_client`	ALF order entry (`pm-alf-gwy`)	`protocol` builds and parses ALF lines; `connection` holds one non-blocking TCP session (`AlfConnection`)	`ai_trader/transport.py`
`calf_client`	CALF market data (`pm-md-gwy`)	`CalfClient` for raw frames; optional cached `MarketState` (top of book, depth, phase, halts). Handles reconnect with backoff, per-`(channel, symbol)` gap detection, `RESUME` repair with de-duplication, and `REF=` tick precision	`calf_spy/client.py`

Two points that are easy to get wrong:

- `calf_client` is the *production* CALF client. The CALF chapter of the Operator Guide documents it for people writing subscribers. `alf_client` is deliberately minimal and has no user-facing chapter.
- `docs/examples/alf/python/alf_client.py` is **not** the `alf_client` package. It is a standalone, interactive example client shipped for protocol learners, and it shares nothing with the package. When changing the ALF wire format, update both.

Prefer these libraries over a hand-rolled socket loop when adding a new in-repo ALF or CALF consumer.

8.3.2 Good first files to read

If you want to understand the runtime quickly, read:

1. `src/edumatcher/engine/main.py`
2. `src/edumatcher/engine/config_loader.py`
3. `src/edumatcher/alf_gwy/gateway.py` — the simplest gateway, and the one Order flow traces end to end
4. `src/edumatcher/models/message.py`
5. `tests/test_*` files closest to the area you plan to change

If you prefer a narrative to a file listing, read Order flow: from keystroke to resting order first — it walks one order through every file above in the order the code touches them.

8.4 Development workflow expectations

8.4.1 Default working style

For most changes, follow this loop:

1. Understand the existing behavior in code and tests
2. Make the smallest complete change that solves the problem
3. Run the narrowest relevant tests first
4. Run the standard quality gate before considering the work done
5. Update documentation if behavior, commands, or configuration changed

8.4.2 Code quality gates

These are the checks a developer is expected to run regularly:

```
make check
make test
```

The Makefile provides wrappers if you prefer shorter commands:

```
make install     # install Python dependencies
make check       # run static lints
make test        # run all tests
make docs-site   # build documentation site
make build       # build Python wheel
```

but the standard way to run all checks and a full build is the `mkbld.sh` script

```
./scripts/mkbld.sh
```

8.4.3 Standards enforced by the repo

- **Formatting**: `black`, line length 88
- **Linting**: `flake8`
- **Typing**: `mypy` in strict mode
- **Testing**: `pytest`
- **Coverage gates**:
  - both `make test` and `scripts/mkbld.sh` enforce the same **80%** threshold (`Makefile`'s `COVERAGE := 80`, mirrored in `mkbld.sh`'s `COVERAGE="80"`)
- **Docs build**: `mkdocs build` should pass after doc changes
- **Shell completion**: every `[tool.poetry.scripts]` entry point must expose a module-level `build_parser() -> argparse.Argumer` with no parameters and no side effects – `main()` calls it too, so there is exactly one place each parser is defined. `tools/gen_completion.py` reads those factories to render `src/edumatcher/completion/pm-completion.{bash,zsh}` (shipped as package data, served by `pm-help --completion bash|zsh`); `tests/test_shell_completion.py` fails both if a command lacks the contract and if the committed scripts drift from what the factories currently produce. After adding a command or changing a flag, subcommand, `choices=` list or help text, run `make completion` and commit the result – see Installation → Shell completion.
- **REST contract**: `spec/rest/openapi.json` is the committed v1 contract, exactly what `pm-api-gwy` serves at `/openapi.json`. `make openapi-check` (part of `make check` and CI) fails, with a diff, when a route, model, parameter or description no longer

matches it. After a *deliberate* REST change, run `make openapi` and commit the result with the code. The diff is where a reviewer decides whether the change is additive. Unlike the message bindings, `scripts/mkbld.sh` never regenerates it, so a build cannot quietly accept an API change. Error codes are held exactly: each route lists the codes it can return in `responses=docs.errors(...)`, a test derives the same set from the code (`tests/api_gateway_error_scan.py`), and a new `error.code` must first be added to `api_gateway/errors.py`. The REST spec's endpoint tables (arguments, success statuses, error codes) are checked against the contract by `tests/test_rest_spec_matches_contract.py`. See Compatibility and deprecation policy.

8.4.4 What to preserve

This repository values:

- exact behavior over speculative abstraction
- deterministic tests over hand-wavy correctness
- documentation that matches the real code
- small, reviewable changes instead of large refactors

If you notice unrelated technical debt while doing a focused task, note it, but do not silently expand the scope of your change.

8.5 Running a minimal system while developing

It is worth keeping a **small live system** available during development. Even when unit tests pass, a real end-to-end run catches startup, wiring, and event flow mistakes early.

This section covers a minimal system on the host. If you are working on one of the web applications instead, the fastest loop is the exchange in containers with the app running locally against it — see The Development Loop.

8.5.1 Where your data directory actually is

Every process — `pm-engine`, `pm-setup`, `pm-config-deploy`, the test suite, all of them — resolves one shared `DATA_DIR` (`edumatcher.config`) and reads/writes everything under it: `ref_data/engine_config.yaml`, the compiled `ref_data/engine_config.json`, `gtc_orders.json`, the audit and stats databases, and so on. Which directory that is depends on how EduMatcher is running, and mixing the two up is a common source of "why doesn't my change do anything" / "why does pytest suddenly fail" confusion:

You are	`DATA_DIR` resolves to
Running from a source checkout (`poetry run ...`, or any test you run with `pytest`)	`<repo>/src/data/` — this is the **default in a dev environment**
Running an installed build (`pipx install edumatcher`, no source checkout)	`~/.local/share/edumatcher`
Either of the above, with `EDUMATCHER_DATA_DIR` set	Whatever that variable points at — always wins

The detection is automatic and based on where `edumatcher/config.py` itself lives on disk (inside a `src/` directory means "source checkout"), not on how you invoke a command. So in this repository's normal poetry workflow, `src/data/ref_data/engine_config.json` is **the** compiled artifact every local process and every test that loads real config reads — there is no separate "test data dir" or "dev data dir" to keep in sync.

Customizing the location: export `EDUMATCHER_DATA_DIR=/some/path` before running any EduMatcher process; every process that reads `DATA_DIR` will follow it consistently as long as the variable stays set for all of them (e.g. in your shell profile, or in a `.env` a launcher script sources). This is the same variable `pm-setup --data-dir` and the containerized `edumatcher.sh` deployments use, so a single override applies everywhere. See the `Runtime configuration` docstring at the top of `src/edumatcher/config.py` for the full priority order, including the host/container-split fallback to `./data` in the current directory.

Why this matters for `pm-config-deploy` / `pm-setup`: these compile a source YAML into `<DATA_DIR>/ref_data/engine_config.json`, and every process refuses to start against a stale or hand-edited copy of that file (it verifies a content digest recorded at compile time). If you change something about the compiled config's shape — add a field to `EngineConfig`, for instance — an artifact compiled by older code will fail that check under the new code with `ArtifactError: compiled config has been modified since it was compiled`. That is not corruption; it means `src/data/ref_data/engine_config.json` in your checkout needs recompiling: run `poetry run pm-config-deploy --example <name>` (or `pm-setup --force`) again so it's rebuilt under the current code. This

is *exactly* the situation a full local `pytest` run will hit right after pulling a change that touches the compiled config's shape — the failures are pointing at `src/data/`, not at the test files.

The three config commands divide up as: **`pm-config-gen`** authors a fresh `engine_config.yaml` from high-level CLI inputs (symbol count, gateways, sessions) when you want a new scenario rather than an edit to an existing one; **`pm-config-deploy`** compiles an authored YAML into the deployed `ref_data/` artifacts; and **`pm-config-show`** prints what is actually deployed, which is the one to reach for when the engine's behaviour and your YAML appear to disagree.

8.5.2 Minimal reference data

EduMatcher uses `engine_config.yaml` for reference data. A one-symbol minimal configuration can look like this:

```
participants:
  - id: TRADER01
    description: First trader
  - id: MM01
    description: Market maker
    role: MARKET_MAKER
  - id: GW_ADMIN
    description: Operator console
    role: ADMIN

symbols:
  AAPL:
    tick_decimals: 2
    last_buy_price: 149.90
    last_sell_price: 150.10
    market_maker_quotes:
      - gateway_id: MM01
        bid_price: 149.90
        ask_price: 150.10
        bid_qty: 500
        ask_qty: 500
        tif: DAY
        quote_id: MM-AAPL-SEED
```

See The Configuration Workflow for the full schema.

8.5.3 Recommended startup order

Start the engine first. All other processes depend on its sockets being bound.

Core background/exchange processes

```
# Terminal 1 - Start the log server
poetry run pm-log-srv

# Terminal 2 — audit log
poetry run pm-audit --terminal

# Terminal 3 — matching engine
poetry run pm-engine --verbose

# Terminal 4 — optional scheduler
poetry run pm-scheduler --now

# Terminal 5 - Statistics
poetry run pm-stats

# Terminal 10 — clearing / P&L
poetry run pm-clearing
```

An easier way to start all basic exchange processes is to use the admin command `pm-opctl-cli` (OPerationall Control Client)

```
pm-opctl-cli start
```

to then check all processes are running

```
pm-opctl-cli list
```

and finally to stop all processes

```
pm-opctl-cli stop
```

Start obersvers

```
# Terminal 9 — live order book
poetry run pm-viewer --symbol AAPL

# Terminal 10 — live all symbol display
poetry run pm-board
```

Start interactive terminals

```
# Terminal 11 — Trader01 terminal
poetry run pm-alf-console --id TRADER01

# Terminal 12 — Trader02 terminal
poetry run pm-alf-console --id TRADER02

# Terminal 8 — operator console
poetry run pm-admin --id GW_ADMIN
```

8.5.4 Inspecting a run after the fact

The processes above are the *live* view. Every store they write also has a read-only query CLI, and reaching for one is almost always faster than opening the SQLite file or grepping a log by hand:

Command	Reads	Reach for it when
`pm-audit-cli`	`data/audit.log`, `data/audit/` (day parts, `.zst` archives)	"what happened on the bus?" — filter by topic, gateway, symbol, time
`pm-stats-cli`	the stats database	OHLCV, trades, order events, order lifecycle, gaps
`pm-clearing-cli`	the clearing database	positions, P&L, and `reconcile` against the raw trade archive
`pm-log-cli`	`pm-log-srv`	operational logs from every process in one place
`pm-index-admin-cli`	`pm-index`	corporate actions, constituent changes, rebalances

Three more that are easy to miss:

- **`pm-mm-bot`** — the market-maker bot. Start one before a demo and the book has resting liquidity to trade against; without it an AI-trader swarm produces a thin, jumpy book. See AI bot traders.
- **`pm-dc-spy`**, **`pm-ralf-spy`**, **`pm-calf-spy`** — tail the drop-copy, RALF and CALF feeds without writing a subscriber. The fastest way to see whether a gateway is emitting what you think it is.
- **`pm-help`** (alias **`pm-man`**) — browses the documentation for any EduMatcher command from the terminal.

8.5.5 Logging levels and pm-log-srv

Every `pm-*` process configures its own logging once at startup, in a `_configure_logging(args)` function in that process's `main.py`. Three things are decided there: the **level** (from `--log-level {CRITICAL,ERROR,WARNING, INFO,DEBUG}`, or `-v`/`-vv`/`-q` if `--log-level` is not given — see the flag table under `pm-engine` for the exact precedence, which is the same in every process),

and the **target** — where the resulting `LogRecords` go, via `--log-target {server,stdout,file}`. The default target, `server`, auto-detects a running `pm-log-srv` with a LALF `HELLO/WELCOME` handshake and, if one answers, attaches a `TcpLogHandler` as the process's *only* logging handler. That handler ships every record — at whatever level the process was started with — over TCP to `pm-log-srv`, so `log.debug(...)` calls throughout the codebase reach the same centralized log you would otherwise read with `pm-log-cli` or `log-gui`, with no separate wiring required.

This has two consequences worth knowing before you go looking for a way around them:

- **There is no environment variable.** The level is a plain CLI argument, resolved once per process at startup. Nothing in this codebase reads a `LOG_LEVEL`-style variable.
- **The level cannot be changed on a running process.** `_configure_logging` runs once in `main()`, before the process's event loop starts, and nothing listens for `SIGHUP` or any other runtime toggle. To see `DEBUG` output from a process, restart it with `--log-level DEBUG` (or `-vv`). `pm-log-srv` itself does not need restarting — it is a passive collector — only the client process whose behaviour you are debugging does.

If you normally bring the stack up with `pm-opctl-cli` rather than starting processes by hand, start (or restart) it with `-d`/`--debug` (alias `-vv`) to apply `--log-level DEBUG` to every process in the profile at once:

```
pm-opctl-cli stop
pm-opctl-cli start -d
```

See `pm-opctl-cli` for the full flag reference.

8.5.6 macOS convenience launcher

For demos or quick manual runs on macOS, use:

```
./tools/launch_all.sh
./tools/launch_all.sh AAPL MSFT
```

`tools/launch_all.sh` opens one Terminal window per process using AppleScript. It is convenient, but it is **macOS-only** and it launches the standard demo layout, not a custom research topology.

8.5.7 Quick signs the system is healthy

You should expect to see:

- engine startup banner with bound ports `5555`, `5556`, and `5557`
- successful gateway authentication
- a visible two-sided book in `pm-viewer` after `MM01` connects
- order acknowledgements and fills flowing back to the submitting gateway

8.6 Verification and test strategy

There are three different kinds of checks in this repository, and they answer different questions.

8.6.1 Normal regression tests

Run these continuously while developing:

```
poetry run pytest -n logical tests/ -m "not perf"
```

or via the master Makefile

```
make test
```

These are the default correctness tests and should remain fast enough for frequent reruns.

8.6.2 Deterministic engine verification

EduMatcher includes a dedicated replay-and-compare verification flow under `tools/`. This is the right choice when you need confidence that the production engine still agrees with the paper-trading oracle.

```
bash tools/verify_matching.sh
```

Useful variants:

```
bash tools/verify_matching.sh --seed 7
bash tools/verify_matching.sh --count 500
bash tools/verify_matching.sh --tolerance 0.01
```

Read Verification before changing this flow. It explains why deterministic replay is hard and how the repository avoids common traps such as clocks, ACK ordering, and persisted GTC state.

8.6.3 Performance tests

Performance tests are intentionally separate from the normal CI path. They measure engine behavior, not the full end-to-end network stack.

```
# Full perf run
poetry run pytest -o addopts='' tests/test_perf.py -v -s -m perf -p no:cov
# or
make test-perf

# Throughput-focused view
poetry run pytest -o addopts='' tests/test_perf.py -v -s -m perf -k max_tps -p no:cov

# Latency-focused view
poetry run pytest -o addopts='' tests/test_perf.py -v -s -m perf -k latency -p no:cov

# Normal test run without perf tests
poetry run pytest tests/ -m "not perf"
```

Important interpretation rule: the performance tests primarily measure **engine processing cost**, not total production wire latency.

8.6.4 When to run which check

Situation	Minimum check
Small logic change in one module	Narrow tests for that area, then `pytest tests/ -m "not perf"`
Message schema or process wiring change	Normal tests + a live minimal-system run
Matching-engine algorithm change	Normal tests + `tools/verify_matching.sh`
Performance-sensitive hot-path change	Normal tests + performance tests
Documentation-only change	`poetry run mkdocs build`

8.7 Helper scripts under `scripts/` and `tools/`

The repo includes useful helper scripts, but not all of them are equally authoritative. In general:

- prefer **Poetry and Makefile commands** for day-to-day work
- use scripts for automation, release flow, or convenience wrappers
- read a script before trusting it in a new CI or release workflow

8.7.1 Core scripts

Script	Use it for	Notes
`scripts/mkbld.sh`	Full local build / validation pipeline	Runs lint/type/tests/build/docs; updates README version line; Exchange Intro PDF build is optional and requires `--intro`

Script	Use it for	Notes
`scripts/mkchlogentry.sh`	Create a new `CHANGELOG.md` release template	Intended before `mkrelease.sh`
`scripts/mkrelease.sh`	Local release workflow from `develop`	Requires `GITHUB_USER`, clean/synced `develop`, existing changelog entry for current `pyproject.toml` version, and already-built `dist/` artifacts
`scripts/mkghrelease.sh`	Publish the GitHub release from `main`	Requires `gh` auth, clean/synced `main`, latest `v*` tag, and release artifacts (wheel, sdist, user-guide bundle, Exchange Intro bundle)
`scripts/mkdocs.sh`	Serve, build, deploy, or clean MkDocs docs	Helpful for docs-only work
`scripts/mkcovupd.sh`	Update the README coverage badge from `coverage.xml`	Secondary helper; inspect output before committing
`scripts/verify_setup.sh`	Smoke-check a local environment	Contains some inherited naming; use with caution
`scripts/docs-contctl.sh`	Run docs in a Podman container	Useful when validating the containerised docs image
`tools/verify_matching.sh`	Deterministic engine verification	Strong confidence check for engine changes
`tools/launch_all.sh`	macOS demo/process launcher	Good for manual demos, not for production orchestration
`tools/gen_completion.py`	Regenerate bash/zsh shell completion for every `pm-*` command	Dev-only (imports `shtab`); run via `make completion` after changing a command's flags

8.8 Documentation workflow

Developer-facing work is not finished until the docs still build cleanly.

8.8.1 Fast documentation loop

```
poetry run mkdocs serve
```

8.8.2 One-shot validation

```
poetry run mkdocs build
```

8.8.3 What to update when behavior changes

If you change:

- **configuration semantics** → update `../../operator-guide/part-2-configure/010-the-configuration-workflow.md`
- **runtime commands or startup behavior** → update `../../operator-guide/part-3-run/010-running-the-exchange.md`
- **gateway commands** → update `../../participant-guide/part-2-orders/010-the-trader-console.md` (and `../../participant` for general gateway concepts)
- **message payloads or topics** → update `../../protocols-and-clients/part-5-message-reference/010-message-reference.md`
- **risk, MM quotes, persistence, or drop copy** → update the corresponding user-guide page
- **developer workflow** → update this page and related developer docs

This repository already has a lot of explanatory documentation. Reuse it rather than duplicating large explanations in new pages.

8.9 Mermaid diagrams in the docs build (cached Pandoc filter)

The PDF docs pipeline (`docs/Makefile`, `docs-design/Makefile`, `docs-exchange-intro/Makefile`) renders every ```` ```mermaid ```` code block to an image via Pandoc's `--filter` mechanism. We use our **own** filter, `scripts/mermaid-filter-cached.js`, instead of the stock `mermaid-filter` npm package installed under `build-tools/node_modules/`.

8.9.1 Why a custom filter

Rendering one diagram means launching `mmdc` (`@mermaid-js/mermaid-cli`), which starts a full headless Chrome instance via Puppeteer — by far the slowest step in a documentation build, repeated once per diagram, on every single build, even though the diagrams themselves rarely change. Building all four PDF variants (A4/B5, light/dark) in parallel (`make -j4`) multiplies that cost further, since they share the same Markdown sources.

`scripts/mermaid-filter-cached.js` adds a render cache on top of the stock filter's behavior: a diagram whose Mermaid source hasn't changed is reused from disk instead of being re-rendered.

8.9.2 Why it isn't just a patch to the installed package

`build-tools/` is entirely git-ignored and is wiped and recreated by `npm install` on every fresh checkout (see the `$(NODE_MODULES_PATH):` targets in the Makefiles). Editing `build-tools/node_modules/.bin/mermaid-filter` (a symlink into `build-tools/node_modules/mermaid` directly would be silently lost the next time dependencies are (re)installed. The cached filter therefore lives as a normal, version-controlled file under `scripts/`, and each Makefile's `MERMAID_FILTER` variable points at it instead of the npm-installed binary.

8.9.3 How the cache works

- **Cache key**: sha256 of the raw Mermaid diagram text, truncated to 12 hex characters — independent of caption, filename, or which document the diagram appears in, so two identical diagrams anywhere share one render.
- **Cache location**: `build-tools/.mermaid-cache/<hash>.<format>` by default, overridable with `MERMAID_FILTER_CACHE_DIR`. This is deliberately **not** the per-build `.mermaid-img` directory the Makefiles pass via `MERMAID_FILTER_LOC` — that directory is `rm -rf`'d at the start of every PDF build, so anything cached there would never survive to be reused.
- **Cache hit**: `mmdc`/Puppeteer is skipped entirely; the previously rendered file is referenced directly from the cache.
- **Cache miss**: rendered via `mmdc` exactly as the stock filter does, then persisted into the cache using a write-to-temp-then-atomic-rename so that concurrent builds (the four `make -j4` PDF variants, which share diagrams) can never observe — or produce — a partially-written cache file.

Caveat: the cache key covers only the diagram text, not rendering options (`MERMAID_FILTER_THEME`/`WIDTH`/`FORMAT`/`SCALE`/…). Changing one of those globally does not invalidate previously cached renders. To force a full re-render, delete `build-tools/.mermaid-ca` (or point `MERMAID_FILTER_CACHE_DIR` somewhere fresh).

CHAPTER 9

The Development Loop

Learning objectives

After reading this page you will understand:

- Why a change in `web-apps/` is invisible after `make up-all`, and the one flag that fixes it
- How to run a web application on your own machine, with hot reload, against the exchange running in containers — and why that works without any configuration
- Exactly what each of the six applications needs from a running backend: which ports, which files, which credentials
- Which of the three loops to reach for, given the change you are making
- The handful of things that will waste an afternoon if you do not know them

9.1 Summary

There are three ways to run EduMatcher while developing it, and the whole chapter is about choosing between them.

	What runs where	Feedback	Use it for
Container loop	Everything in containers	~1–3 min per web-app change	Verifying what you will ship; anything touching the Dockerfile, compose files or the entrypoint
Hybrid loop	Exchange in containers, one web app on the host	Instant — Vite HMR, `tsx watch`	Almost all front-end and bridge work
Local loop	Everything on the host (`pm-setup`, `pm-opctl-cli`)	Instant	Python work on the engine, gateways and CLIs

The hybrid loop is the one to reach for by default when working on `web-apps/`, and it needs no special build of anything:

```
cd deployment/docker && make up-all              # the exchange, in containers
eval "$(make -s dev-env GUI=terminal-gui)"       # the two values that cannot be defaulted
cd ../../web-apps/terminal-gui && make dev       # Vite on 8190, bridge on 5190
```

That works because **every bridge defaults its connect targets to `127.0.0.1`**, and `127.0.0.1` is exactly where the container stack publishes its ports. The container and the host see the same addresses.

9.2 Getting started: from a fresh clone

Everything below assumes this has been done once.

Prerequisites

Tool	Needed for	Notes
Podman or Docker (with Compose)	The container stack	Podman is preferred when both are installed; `make info` in `deployment/docker` shows what was detected
GNU make	Every workflow	
Poetry and Python	`make build` builds the backend wheel from your checkout	See Development Practice
Node.js **22.13 or newer**, with npm	Hybrid loop, `make install`, `make test` in `web-apps/<app>`	The images build on Node 22. Most apps accept Node 20, but `trader-gui` needs 22 and `dash-gui` 22.13, so use one version for all of them. Install it with nvm or fnm

First run

```
cd deployment/docker
make build            # backend image, from a wheel built from your checkout
make up-all           # the exchange and five GUIs; builds any missing GUI image
```

The first run is slow (every GUI image runs `npm ci` and a Vite build) and needs network access to the npm and PyPI registries. Later starts take seconds. When it finishes, the GUIs answer on http://localhost:8090 to http://localhost:8095 (`config-gui` on 8092 only with `CONFIG_GUI=1`). `make status` shows one row per backend process.

Which Makefile? For development use `deployment/docker` (`make up-all`). The `make up` in `web-apps/` starts only the GUI containers against a backend that is already running elsewhere, such as a VM. It does not include `config-gui`.

Before you push web-app changes. The release gate (`mkbld.sh`) runs these for every app, so run them in the app you touched:

```
cd web-apps/<app>
make typecheck && make lint && make test
```

On Linux with Docker, GUI containers reach the host through a `docker-compose.linux.yml` overlay that the app Makefiles add automatically. With Podman, override the `*_HOST` variables to `host.containers.internal`. The hybrid loop below is unaffected, because it uses `127.0.0.1`.

9.3 Part 1 — Why `make up-all` does not pick up your change

`up-all` runs `compose up -d`, deliberately *without* `--build`. Compose only builds an image that does not exist yet, so an image tagged `edumatcher-terminal-gui:latest` from an hour ago is reused as-is. Your change is in the working tree, not in the image, and nothing warns you.

The reason it is not simply `--build` by default: that re-evaluates six build contexts on every start (the six GUIs plus the backend), which turns a three-second start into a multi-minute one. Building is a separate decision from starting.

So the loop is *build, then start*:

```
make build-guis GUI=terminal-gui     # rebuild one image
make up-all                          # recreate the containers whose image changed
```

or, in one step:

```
make up-all BUILD=1 GUI=terminal-gui
make up-all BUILD=1                  # all six
```

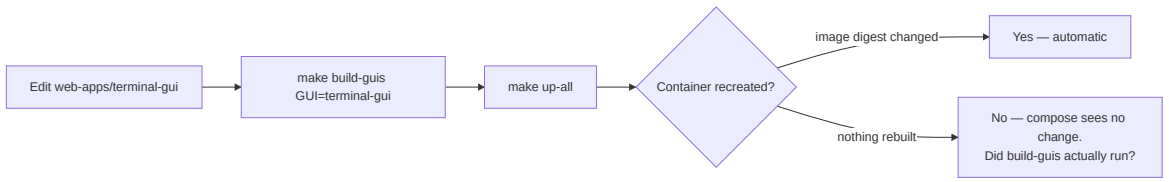
`make build-guis` with no `GUI=` rebuilds all six. It builds through `compose.yaml -f compose.guis.yaml -f compose.config-gui.yaml`, so config-gui is always buildable here whether or not you start it with `CONFIG_GUI=1`.

For a **backend** change the equivalent is `make build`, which rebuilds the wheel from the checkout first:

```
cd deployment/docker
make build && make up-all
```

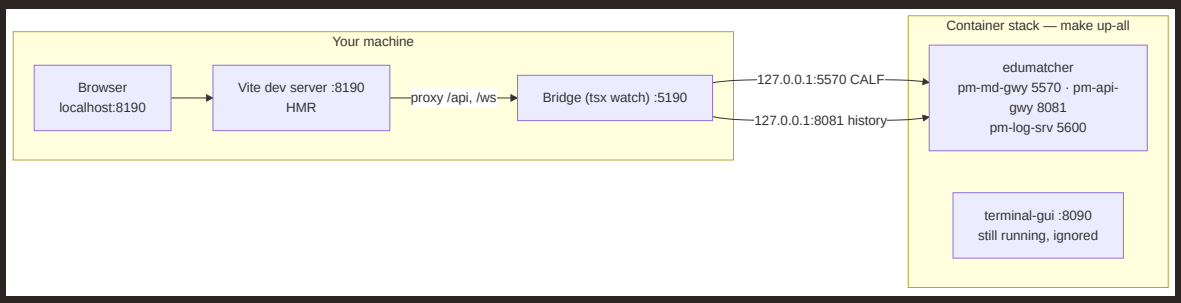
Check the wheel line

`make build` must print `==> building wheel from the repository checkout` followed by `Installing local wheel: ....` If it installed from PyPI instead, you have been testing the released code, not yours. See Build time versus run time.



9.4 Part 2 — The hybrid loop, and why it needs no wiring

Run the exchange in containers and one web application directly on your machine. You get Vite hot module replacement in the browser and `tsx watch` restarting the bridge on save, against a real exchange with real market data.



Three facts make this work with no configuration:

1. **Each app's Vite config already proxies to its own bridge.** `/api` and `/ws` on the dev server are forwarded to `127.0.0.1:<bridge port>`, so the browser only ever talks to one origin and there is no CORS question.
2. **Each bridge defaults every connect target to `127.0.0.1`** — `CALF_HOST`, `LOG_SRV_HOST`, `API_GATEWAY_URL`, all of them.
3. **The stack publishes to `127.0.0.1`** by default (`BIND_ADDR`). The two halves meet without either being told about the other.

The containerised GUI keeps running on 8090–8095 the whole time. That is useful rather than confusing: it is your reference. Compare `localhost:8190` with `localhost:8090` and you are comparing your change against what is currently shipped.

9.4.1 `make dev-env`

Two things genuinely cannot be defaulted: the read-only API key, which is generated per engine configuration, and the paths into this stack's `./data`. `make dev-env` reads them out of the running backend and prints them as `export` lines, so they can be evaluated straight into your shell:

```
$ cd deployment/docker
$ make -s dev-env GUI=terminal-gui
# terminal-gui: web on 8190, bridge on 5190 (Vite proxies /api and /ws)
export API_GATEWAY_URL=http://127.0.0.1:8081
export PM_TERMINAL_API_KEY=key-readonly-2kqhcbckg7rk5mkdf6c3ldjd9hbc8ve5

$ eval "$(make -s dev-env GUI=terminal-gui)"
```

Warnings — a missing credential, an unpublished port, a `log.db` that does not exist yet — go to **stderr**, so they reach you on the terminal without ending up inside the `eval`.

Re-run it after `make up-all CONFIG=<other>`: the read-only key is different in every bundled configuration, and a stale key gives you a live order book with empty history panels and a 401 in the bridge log.

9.4.2 Ports at a glance

Application	Dev web	Dev bridge	Container	Needs from the backend
terminal-gui	8190	5190	8090	5570 CALF, 8081 history + read-only key, 5600 log ingest
log-gui	8191	5191	8091	5601/5602 (needs `ZMQ=1`), `data/log.db`
trader-gui	8193	—	8093	8080 only, proxied by Vite
config-gui	8192	5192	8092	nothing
book-gui	8194	5194	8094	8081 market data, history + read-only key, 5600 log ingest
dash-gui	8195	5195	8095	5611 `pm-monitor` (needs `ZMQ=1`), `data/metrics.db`, the deployed configuration, 5600 log ingest

The dev ports are deliberately `81xx`/`51xx` against the containers' `80xx`, so both can run at once.

9.5 Part 3 — Per-application recipes

Each app needs `make install` once (and again after a dependency change).

9.5.1 terminal-gui

```
cd deployment/docker && make up-all
eval "$(make -s dev-env GUI=terminal-gui)"
cd ../../web-apps/terminal-gui
make install          # first time only
make dev              # bridge + web together
```

Open http://localhost:8190.

`make dev` runs both halves. `make dev-web` and `make dev-bridge` run one each, which is what you want when you are attaching a debugger to the bridge or when the bridge is stable and only the React app is changing.

What it connects to: `pm-md-gwy` on 5570 for the live CALF feed, `pm-api-gwy` on **8081** for history, and `pm-log-srv` on 5600 for its own logging uplink. All three are always published — no `ZMQ=1` needed.

Symptom	Cause
Live book, empty history	`PM_TERMINAL_API_KEY` unset or stale. Re-run `dev-env`
History 401s	The key is valid on 8081 only; `API_GATEWAY_URL` points at 8080
`calf: RECONNECTING`	5570 not published, or the backend is not running

The read-only credential is the one whose `gateway_id` is `null`, issued on the `dashboards` instance. Nothing else in the system needs it, and it is never sent to the browser.

9.5.2 log-gui

```
cd deployment/docker && make up-all ZMQ=1
eval "$(make -s dev-env GUI=log-gui)"
cd ../../web-apps/log-gui && make install && make dev
```

Open http://localhost:8191.

`ZMQ=1` is required here, as for dash-gui. The log bridge subscribes to `pm-log-srv`'s LALF-PS sockets on 5601/5602, and those are ZeroMQ sockets published only by the ZMQ overlay. Without it you get the history views — which read `log.db` directly — and no live stream. `dev-env` checks for this and says so.

`dev-env` also exports two paths:

- `LOG_DB_PATH` → the stack's `data/log.db`, the same file `pm-log-srv` writes. The bridge opens it read-only.
- `ACK_STORE_PATH` → `data/log-ui-acks.db`, the bridge's own store. In the container this is a named volume; on the host it sits beside `log.db`.

If `log.db` does not exist yet, the exchange has not logged anything — `pm-log-srv` creates it on the first entry.

9.5.3 trader-gui

```
cd deployment/docker && make up-all
cd ../../web-apps/trader-gui && make install && make dev
```

Open http://localhost:8193. **No `dev-env` needed**: trader-gui has no bridge, and `vite.config.ts` proxies `/api` — REST and the WebSocket upgrades under `/api/v1/ws/` (`private`, `market-data`, `admin/monitor`) — straight to `http://localhost:8080`, the `desk` instance. Trading credentials are entered in the browser, so there is no key to inject.

That also makes it the one app whose dev server is a true drop-in for the container: the only difference is who serves the static bundle.

9.5.4 config-gui

```
cd web-apps/config-gui && make install && make dev
```

Open http://localhost:8192. **The backend does not need to be running at all.** config-gui is a standalone authoring tool for `engine_config.yaml`; its Fastify server on 5192 shells out to the Python `pm-setup` tooling and talks to no exchange. That is also why it is opt-in (`CONFIG_GUI=1`) in the container stack.

Its dev server needs the Python side available on your `PATH` — `npm run verify:python` checks that.

9.5.5 dash-gui

```
cd deployment/docker && make up-all ZMQ=1
eval "$(make -s dev-env GUI=dash-gui)"
cd ../../web-apps/dash-gui && make install && make dev
```

Open http://localhost:8195.

`ZMQ=1` is required, as for log-gui: the bridge subscribes to `pm-monitor`'s snapshot feed on 5611, which only the ZMQ overlay publishes on the host. Without it you get the history, read from `metrics.db`, but no live figures, and the banner says pm-monitor is down. `dev-env` checks for this and says so.

`dev-env` exports `METRICS_DB_PATH` and `CONFIG_ARTIFACT_PATH` (the stack's `data/metrics.db` and `data/ref_data/engine_config.json`, both opened read-only), `MONITOR_HOST`, and the log-gui and book-gui addresses for the drawers' links. `make test-e2e` at the repository root runs the built page in headless Chrome against a real exchange — see Operator Dashboard for what each part shows.

9.6 Part 4 — Choosing a loop

The change you are making	Loop	Command
React component, styling, a view	Hybrid	`make dev` in the app
Bridge logic, a protocol client under `packages/`	Hybrid	`make dev`; `tsx watch` restarts on save
Anything in `src/edumatcher/`	Local, then container	`poetry run pm-...`; then `make build && make up-all`
A Dockerfile, compose file or `entrypoint.sh`	Container	`make build-guis` / `make build`, then `make up-all`
Ports, published addresses, bind hosts	Container	Only the container exercises the namespace
Anything before a release	Container	It is what CI builds and users install

The rule behind the table: **the hybrid loop is faster but does not exercise the container topology.** A change that is invisible to the network — a component, a query, a formatter — is safe to develop hybrid and verify once in containers at the end. A change to how processes find each other has to be tested where the namespaces are real, because the hybrid loop routes everything over host loopback and will happily hide a bind-address mistake. See The two planes and their defaults.

9.6.1 Editing shared packages

Each app is an npm workspace root, and the Vite configs alias `@edumatcher/<pkg>` to the package's `src/` rather than its build output. A change in `packages/calf-protocol/` therefore hot-reloads in the browser with no build step — but only for the *web* half. The bridge imports the compiled package, so `tsx watch` picks it up on save without a rebuild too, while a production `make build` does compile them.

9.7 Part 5 — Things that will bite

Symptom	Cause
A web-app change does nothing after `make up-all`	No `--build`. Use `make up-all BUILD=1 GUI=<app>`
`make build-guis` succeeds, the browser still shows the old build	The image was rebuilt but the container was not recreated. Run `make up-all` afterwards
`EADDRINUSE` on 8190/8191/8193	A previous `make dev` is still running. The container ports are 80xx and never collide with these
Empty history panels in the hybrid terminal	Stale `PM_TERMINAL_API_KEY` after switching configuration. Re-run `dev-env`
No live entries in the hybrid log viewer	Started without `ZMQ=1`, so 5601/5602 are not published
`dev-env` prints a warning and no exports	The backend container is not running, or has not deployed its configuration yet
Hybrid app works, containerised one does not	Almost always a bind address or a hostname: the hybrid path uses `127.0.0.1`, the container path uses the service name `edumatcher`. This is exactly the class of bug the hybrid loop cannot see
`make dev` fails on a missing module	`make install` has not run since a dependency changed
Backend change has no effect	The image was built from PyPI, not your checkout
`pytest` fails with `ArtifactError: compiled config has been modified since it was compiled`	Your checkout's data directory is `src/data/`, not `~/.local/share/edumatcher/` — its compiled `ref_data/engine_config.json` is stale relative to the code you just pulled. Recompile it: `poetry run pm-config-deploy --example <name>` (or `pm-setup --force`). See Where your data directory actually is

Two exchanges, one set of ports

`deployment/curl/` uses the same container names and host ports as `deployment/docker/`. If a released install is running, the dev stack refuses to start rather than attaching to it — but a *hybrid* app has no such guard, and will happily connect to whichever exchange owns 5570. `./edumatcher.sh stop` in `~/.edumatcher`, or `make mounts` to see which data directory is really behind the container you are talking to.

9.8 Reference

Command	Where	What it does
`make up-all`	`deployment/docker`	Start the exchange and the GUIs
`make up-all ZMQ=1`	`deployment/docker`	… and publish the ZeroMQ bus (needed by hybrid log-gui)
`make build-guis [GUI=<app>]`	`deployment/docker`	Rebuild web-app images
`make up-all BUILD=1 [GUI=<app>]`	`deployment/docker`	Rebuild, then start
`make build`	`deployment/docker`	Rebuild the backend image from a fresh local wheel
`make dev-env GUI=<app>`	`deployment/docker`	Print the environment for a hybrid run
`make mounts`	`deployment/docker`	Which directory is behind each container path
`make install`	`web-apps/<app>`	npm workspace install
`make dev`	`web-apps/<app>`	Dev server(s) with hot reload
`make dev-web` / `dev-bridge`	`web-apps/<app>`	One half only
`make test` / `typecheck`	`web-apps/<app>`	Vitest / TypeScript

Related: Container and Network Setup for the deployed topology, the release pipeline and the network model; Development Practice for the Python side.

CHAPTER 10

Deterministic Verification of the Matching Engine

Learning objectives

After reading this page you will understand:

- Why a matching engine is hard to verify deterministically — and what makes it different from testing a pure function
- What "paper trading" means in the context of algorithmic verification, and why it produces a reliable oracle
- How each of the four verification tools works and how they interact
- The specific traps that real-time clocks, UUIDs, and distributed state set for anyone who wants to replay a set of orders and compare results
- How the toolset sidesteps those traps — and where residual assumptions remain

10.1 Why verification is hard

10.1.1 The matching engine is not a pure function

A pure function always returns the same output for the same input. A matching engine is **not** pure:

- Orders carry UUIDs generated at submission time — two runs produce different IDs.
- The engine tracks wall-clock timestamps for event ordering and expiry.
- `TIF=GTC` (Good Till Cancelled) orders are saved to disk; a second run inherits the state of the first unless that file is deleted.
- The engine runs as a separate OS process communicating over ZeroMQ; network scheduling and OS scheduling can re-order message delivery.
- STOP and STOP_LIMIT orders only trigger when the engine's internal `last_price` crosses the stop level — that internal price is built up incrementally from the same trade sequence, so a single out-of-order delivery poisons every subsequent STOP evaluation.

Any of these effects can cause two "identical" order streams to produce different books, even if the matching algorithm is perfectly correct.

10.1.2 Replay vs. re-simulation

There are two ways to verify a matching engine:

Approach	How	Risk
Replay	Record the exact bytes sent to the live engine, re-send them later, compare	Message timing; GTC state; clock drift
Re-simulation	Run the same orders through a parallel implementation of the algorithm	The parallel implementation may have its own bugs

EduMatcher uses a **hybrid**: orders are generated once as a FIX text file (the single source of truth), then played through two separate execution paths:

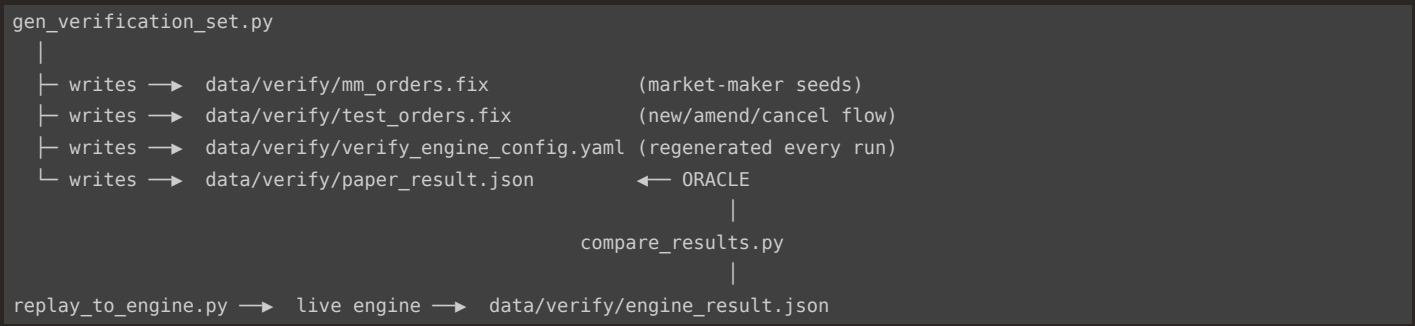
1. **Paper trader** — runs the reference matcher (`tests/reference_matcher.py`) in-process: no ZMQ, no clocks, no GTC persistence. This is the oracle. The reference is a second, deliberately naive implementation written from the documented rules (plain lists re-scanned on every match, no heaps, no indexes), so the comparison is a true re-simulation. Until 2026-10 the paper trader was `OrderBook` itself, which made the comparison blind to any matching bug: both sides ran the same code.
2. **Live engine** — receives the same orders over ZMQ, runs the full production path, then exposes book state via a snapshot API.

If the two books agree at price-level granularity, the engine is verified.

What this does not verify

The toolset compares **resting book state** and **last trade prices**. It does not verify fill attribution (which gateway received which fill), the order of individual trade events, or P&L computations. Those require additional test fixtures (see `tests/test_clearing_ticker_gateway.py`).

10.2 The verification flow



`tools/verify_matching.sh` orchestrates all five steps in sequence.

10.3 Tool 1 — `gen_verification_set.py`

Location: `tools/gen_verification_set.py` **Role:** Generate the FIX input files and run the paper trade to produce the oracle result.

10.3.1 What it builds

The generator creates two FIX files, one YAML config, and one JSON file:

File	Contents
`mm_orders.fix`	56 market-maker limit orders that seed initial liquidity on four symbols
`test_orders.fix`	N lines (default 10 000): ~70% new single-leg orders, ~30% `AMEND`/`CANCEL` against something already resting
`verify_engine_config.yaml`	The engine config this run has to be replayed against — regenerated every time (see The verification config)
`paper_result.json`	Final book state for each symbol after processing all orders in-process

10.3.2 Market-maker seed orders

Each symbol receives 5 bid levels and 5 ask levels centred on a reference mid-price, plus extra depth orders to give each level more than one order in the queue:

Symbol	Mid	Half-spread	Bid L1	Ask L1
AAPL	150.00	0.25	149.75	150.25
AMAZ	180.00	0.25	179.75	180.25
MSFT	420.00	0.50	419.50	420.50
GOOG	160.00	0.25	159.75	160.25

MSFT uses a wider half-spread (0.50) because at higher prices a sub-cent spread is unrealistic — this also exercises the stop-price arithmetic with a different scale.

A seed block looks like this in `mm_orders.fix`:

```
# --- AAPL market-maker seed ---
NEW|ID=mm-AAPL-000|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=500|PRICE=149.75|TIF=DAY
NEW|ID=mm-AAPL-001|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=500|PRICE=150.25|TIF=DAY
NEW|ID=mm-AAPL-002|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=300|PRICE=149.50|TIF=DAY
NEW|ID=mm-AAPL-003|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=300|PRICE=150.50|TIF=DAY
...
```

Every `NEW` line names its own order with an `ID=` field — readable, `mm-<symbol>-NNN` ids for the seed orders — so that a later `AMEND` or `CANCEL` line can refer back to it (see Problem 1 below).

Comment lines (starting with `#`) are silently skipped by both the paper trader and the replay client.

10.3.3 Random test orders: new, amend, or cancel

Each of the N test lines (`--count`, default 10 000) does one of two things:

- **70% of the time** (or whenever nothing is resting yet) it submits a brand new order, from the same weighted distribution of order types as before:

Type	Weight	Notes
`LIMIT`	35%	Price chosen near or away from market
`MARKET`	22%	No price field — sweeps whatever is resting
`ICEBERG`	13%	Large total qty with small `VISIBLE` slice
`FOK`	10%	Fill-or-Kill — cancels if not immediately fully filled
`IOC`	10%	Immediate-or-Cancel — partial fill allowed
`STOP`	5%	Triggers when `last_price` crosses the stop level
`STOP_LIMIT`	5%	Stop that converts to a limit, not a market

  60% of limit-family orders are "near market" (price within 3 half-spreads of mid), making a high fill rate likely and stress-testing deep-book mechanics.

- **30% of the time** (`AMEND_SHARE`), it instead acts on an order that is resting *right now* — picked at random from the live set the generator is tracking as it paper-trades — and emits an `AMEND` or `CANCEL` line against it instead of a new `NEW`:

Outcome	Weight	What it exercises
Reprice	30%	`amend_order`'s re-queue path; see "aggressive" below
Reduce quantity	20%	A reduction keeps time priority
Increase quantity	20%	An increase loses time priority
Reprice **and** resize	10%	Both of the above together
Cancel	20%	Removes the order; frees it from the live set

Only `LIMIT` and `ICEBERG` orders ever rest long enough to be amendable — a `MARKET`/`FOK`/`IOC` order never rests, and a pending `STOP` lives in a separate heap the amend path does not reach — so only those two types are ever chosen as an amend/cancel target.

Of the reprices, 35% deliberately land *through* the opposite side's best price rather than nudging by a cent or two. That turns the amend marketable: the engine pulls the re-queued order back out and runs it through matching instead of just re-resting it — cancel/replace semantics, so an amend can never leave the book crossed. The other 65% are small nudges (±0.01 to ±0.03) that keep the order resting and exercise re-queueing and the priority rules instead.

A typical block in `test_orders.fix`, showing one of each kind:

```
NEW|ID=3f7a1c9b2e4d6801b3f5a9c0d1e2f7a4|SYM=AAPL|SIDE=SELL|TYPE=MARKET|QTY=500
NEW|ID=9c1e4a7f0b3d6289c4e7a0f3b6d9e2c5|SYM=GOOG|SIDE=BUY|TYPE=LIMIT|QTY=250|PRICE=159.23
NEW|ID=0a2d5c8f1b4e7390d5f8b1e4a7c0f3d6|SYM=MSFT|SIDE=BUY|TYPE=ICEBERG|QTY=900|PRICE=419.03|VISIBLE=50
NEW|ID=6e9f2b5c8a1d4073e8b1f4c7a0d3e6f9|SYM=AMAZ|SIDE=SELL|TYPE=STOP|QTY=450|STOP=179.60
NEW|ID=2b5e8c1f4a7d0396f1c4e7b0a3d6f9c2|SYM=GOOG|SIDE=SELL|TYPE=STOP_LIMIT|QTY=200|STOP=159.50|PRICE=159.25
AMEND|ID=9c1e4a7f0b3d6289c4e7a0f3b6d9e2c5|PRICE=159.18
CANCEL|ID=0a2d5c8f1b4e7390d5f8b1e4a7c0f3d6
```

Order ids are random 128-bit hex, generated from the same `--seed`-ed RNG as everything else — not `Order.create()`'s own UUID (more on this under Problem 1).

10.3.4 The paper trader

After writing both FIX files, the generator processes **the same text** through one `ReferenceMatcher` per symbol, without any networking. Simplified:

```
books = {sym: ReferenceMatcher() for sym in SYMBOLS}
for line in mm_lines + test_lines:
    verb, kv = parse_line(line)
    if verb == "NEW":
        books[symbol_of(kv)].submit(reference_order(parse_fix_line(line)))
    elif verb == "CANCEL":
        books[symbol].cancel(kv["ID"])
    elif verb == "AMEND":
        books[symbol].amend(kv["ID"], price=..., qty=...)
```

The reference's `amend` also re-matches a marketable amend, as `Engine._handle_amend` does (cancel/replace), so nothing of the engine's amend logic is copied into the paper trader. `reference_snapshot()` reduces each book to the same price-level shape as the engine's snapshot (an iceberg counts its shown slice).

`submit()`, `cancel()` and `amend()` all return synchronously. There are no sockets, no threads, no timers. The result is deterministic for a given seed — generation and paper-trading happen in the *same* pass, because an amendment has to name an order that is resting at that exact point in the stream, and only the book being built knows which orders those are.

10.3.5 Determinism is guaranteed by the seed

The `--seed` flag is passed directly to `random.Random`. The same seed always produces the same FIX files — new orders, amendments and cancellations alike — which always produces the same paper result:

```
# Reproducible run
poetry run python tools/gen_verification_set.py --seed 42 --count 1000

# Different population, still reproducible
poetry run python tools/gen_verification_set.py --seed 7 --count 500
```

The seed is not stored in the output files. If you lose track of which seed was used to produce a `paper_result.json`, regenerate it — the FIX files are the authoritative input.

10.3.6 Typical output

```
[GEN] Generating and paper-trading 1000 line(s) …
[GEN] Wrote 56 MM orders  → data/verify/mm_orders.fix
[GEN] Wrote engine config    → data/verify/verify_engine_config.yaml
[GEN] Wrote 1000 test lines  → data/verify/test_orders.fix
      769 new, 238 amend, 49 cancel → 879 trades
  AAPL    bids=12 levels  asks=14 levels  last=150.22
  AMAZ    bids=8 levels  asks=17 levels  last=180.30
  MSFT    bids=16 levels  asks=13 levels  last=418.83
  GOOG    bids=9 levels  asks=25 levels  last=160.07
[GEN] Saved paper result     → data/verify/paper_result.json
[GEN] Done — run replay_to_engine.py to get the engine result.
```

(Real numbers for `--seed 42 --count 1000`, reproduced exactly on every run with that seed.) There is no "skipped" count any more — the four `#` comment header lines in `mm_orders.fix` are silently skipped while building the MM book, and `replay_to_engine.py`'s own parse-error counter (below) is what surfaces them later.

10.4 Tool 2 — `replay_to_engine.py`

Location: `tools/replay_to_engine.py` **Role:** Connect to a running engine as gateway `VERIFY01`, send every order from the FIX files (waiting for each ACK), then request and save book snapshots.

10.4.1 Prerequisites

`gen_verification_set.py` writes a fresh `verify_engine_config.yaml` on every run (admitting only gateway `VERIFY01`, same four symbols) — it still has to be **deployed** before the engine will use it, and then the engine started:

```
poetry run pm-config-deploy data/verify/verify_engine_config.yaml
poetry run pm-engine
```

No process takes a config path as an argument, so this deploy step is what makes the just-generated config the one the engine actually loads; skip it and the engine keeps whatever configuration was deployed before.

10.4.2 Protocol

The replay client is a thin ZeroMQ wrapper:

```
                   PUSH ──────────────────►  Engine PULL
Client (VERIFY01)                           Engine
                   SUB  ◄──────────────────  Engine PUB
```

Before sending any orders it authenticates: it PUSHes a `gateway_connect` for `VERIFY01` and waits (up to 3 000 ms) on the SUB socket for `system.gateway_auth.VERIFY01` to come back accepted.

Then, for each dataset line:

1. **`NEW`** — encode the `Order` as `make_order_new_msg(order.to_dict())`, `push_sock.send_multipart(frames)`, then poll SUB for `order.ack.VERIFY01` (up to 2 000 ms) and match its `order_id` against the order just sent.
2. **`AMEND`** — send `make_order_amend_msg(order_id, "VERIFY01", price=…, qty=…)` and wait for either `order.amended.VERIFY01` (accepted) or an `order.ack.VERIFY01` carrying `accepted=False` (refused) for that `order_id`.
3. **`CANCEL`** — send `make_order_cancel_msg(order_id, "VERIFY01")` and wait the same way on `order.cancelled.VERIFY01`.

Only once a line's confirmation arrives does the client send the next line.

This **synchronous send-and-wait loop** is the single most important design choice in the replay tool. Without it, the engine's internal message queue can reorder lines, breaking STOP trigger sequences and letting an `AMEND` or `CANCEL` overtake the line it targets — either way making the result non-deterministic.

Why ACK matching matters for STOPs

A STOP order on AAPL triggers when `last_price` crosses its stop level. If AAPL-LIMIT orders are sent ahead of the stop without waiting for their ACKs, the engine may process them in a different order than the paper trader did. The `last_price` diverges, and the STOP may never trigger (or trigger too early). The ACK loop enforces the same ordering in

both paths.

10.4.3 Snapshot collection

After all orders are sent, the client pauses 0.5 s (configurable via `DRAIN_PAUSE_S`) to let the engine finish any async work, then sends a `book.snapshot_request` message for each symbol:

```
push_sock.send_multipart(make_book_snapshot_request_msg("AAPL"))
# Engine publishes the snapshot to topic  book.AAPL
snap = wait_for_topic("book.AAPL", timeout_ms=5000)
```

The snapshot is normalised to the same `{bids, asks, last_price, ...}` shape used by the paper result, then saved to `data/verify/engine_result.json`.

10.4.4 Typical output

(Real output for the `--seed 42 --count 1000` dataset generated above, with the engine running with no competing resting orders — a "rej" line for `NEW`/`AMEND`/`CANCEL` all count toward the same `rejected` tally.)

```
[REPLAY] Gateway VERIFY01 authenticated.
[REPLAY] … 500 lines sent (500 acc, 0 rej)
[REPLAY] … 1000 lines sent (1000 acc, 0 rej)
[REPLAY] Sent 1056 lines in 0.59s  (1056 accepted, 0 rejected, 4 parse errors)
[REPLAY] Pausing 0.5s for engine to drain …
[REPLAY] Requesting book snapshots …
  AAPL    bids=12 levels  asks=14 levels  last=150.22
  AMAZ    bids=8 levels  asks=17 levels  last=180.3
  MSFT    bids=16 levels  asks=13 levels  last=418.83
  GOOG    bids=9 levels  asks=25 levels  last=160.07
[REPLAY] Saved engine result → data/verify/engine_result.json
[REPLAY] Done — run compare_results.py to verify.
```

Progress prints every 500 lines (not every 100), and the "4 parse errors" here are the same four `#` comment lines in `mm_orders.fix` seen in Tool 1 — this tool's own parser skips them too, but the summary line is the only place that counts them. Same `--seed 42 --count 1000` run throughout, so the book state (and `compare_results.py`'s verdict below) match Tool 1's example exactly.

10.5 Tool 3 — `compare_results.py`

Location: `tools/compare_results.py` **Role:** Load both JSON files, compare them field by field, and print a PASS or FAIL verdict with a full diff.

10.5.1 What is compared

For each symbol, the comparison checks:

Field	Comparison
Bid levels	Price and total visible qty at each level, sorted high→low
Ask levels	Price and total visible qty at each level, sorted low→high
`last_price`	Last trade price (both sides)
`last_buy_price`	Last price where the aggressor was a BUY
`last_sell_price`	Last price where the aggressor was a SELL

10.5.2 What is deliberately not compared

- **Order IDs** — the paper trader uses `gateway_id="PAPER01"` and the engine uses `"VERIFY01"`. Even if the IDs matched, comparing them would be wrong because individual orders can be partially filled and recombined differently.
- **Number of resting orders per level** — only total visible qty matters for the book state. The paper trader and engine may have placed depth orders at the same price as separate entries; the snapshot already aggregates them.

- **Fill counts / trade events** — these are in-flight and not captured in the snapshot.

10.5.3 Tolerance flag

For quantities, the default is an exact match. If floating-point rounding in the engine produces a minor deviation, use `--tolerance`:

```
# Allow up to 0.5% quantity discrepancy per level
poetry run python tools/compare_results.py --tolerance 0.005
```

Two other flags exist for ad-hoc use: `--paper FILE`/`--engine FILE` point at a different pair of result files, and `--symbols SYM [SYM …]` restricts the comparison to a subset of the four.

10.5.4 Reading the output

Before the per-symbol breakdown, the tool prints a compact side-by-side table — bid/ask level counts and `last_price` for both sides, one row per symbol, with a ✓/✗ glyph summarising each row. Then, for each symbol:

A passing run (real output, same `--seed 42 --count 1000` dataset as the Tool 1 and Tool 2 examples above):

```
── AAPL ──────────────────────────────────
  ✓  PASS  — 12 bid level(s), 14 ask level(s), last=150.22

── AMAZ ──────────────────────────────────
  ✓  PASS  — 8 bid level(s), 17 ask level(s), last=180.3

══  RESULT: PASS  ══
```

A failing run with a qty mismatch looks like this (illustrative — not from a real failing dataset):

```
── AAPL ──────────────────────────────────
  ✗  BID[2] qty @ 149.50: paper=450, engine=400
  ✗  last_price: paper=149.55, engine=149.75

══  RESULT: FAIL  ══
```

Note the per-symbol headers use light dashes (—), not the heavy ═══ that `verify_matching.sh`'s own step headers use (see below) — the two scripts format their output independently.

Exit code is `0` on PASS and `1` on FAIL, making it composable in CI pipelines.

10.6 Tool 4 — `verify_matching.sh`

Location: `tools/verify_matching.sh` **Role:** Orchestrate all steps in a single shell command.

```
./tools/verify_matching.sh
```

The script performs five steps:

1. **Generate** — run `gen_verification_set.py` to produce the FIX files, the (re)generated `verify_engine_config.yaml`, and the oracle.
2. **Start engine** — resolve the engine's real data directory, delete any stale GTC/book-stats persistence files in *that* directory, deploy `verify_engine_config.yaml` as the active configuration (`pm-config-deploy`), then launch `pm-engine` in the background and wait 2 s for it to bind.
3. **Replay** — run `replay_to_engine.py` against the running engine.
4. **Stop engine** — send SIGINT and wait for it to exit.
5. **Compare** — run `compare_results.py`.

10.6.1 Command-line flags

```
./tools/verify_matching.sh                    # seed=42, 10 000 lines
./tools/verify_matching.sh --seed 7           # reproducible with a different population
./tools/verify_matching.sh --count 200        # faster smoke test
./tools/verify_matching.sh --skip-gen         # reuse existing .fix files (re-run engine only)
./tools/verify_matching.sh --tolerance 0.005  # pass 0.5% qty tolerance to compare
./tools/verify_matching.sh --pull tcp://localhost:5555 --pub tcp://localhost:5556
                                               # non-default engine addresses
```

`--pull`/`--pub` are passed straight through to `replay_to_engine.py`; both default to the engine's real PULL/PUB addresses (5555/5556).

10.6.2 Clean-state guarantee

Stale GTC state is the most common cause of spurious failures. Naively deleting `data/gtc_orders.json` et al. relative to the repository root used to be exactly that kind of bug: in a source checkout the engine's real data directory is `src/data/`, not `data/` (see Where your data directory actually is) — so the old version of this step silently cleaned a directory the engine never read, and every run "started clean" while actually inheriting the previous run's resting orders. The script now asks the code where the data directory really is, rather than assuming:

```
DATA_DIR="$(poetry run python -c 'from edumatcher.config import DATA_DIR; print(DATA_DIR)')"
rm -f "$DATA_DIR/gtc_orders.json" \
      "$DATA_DIR/gtc_combos.json" \
      "$DATA_DIR/book_stats.json"
```

If you are running the script inside a CI job that also runs the main test suite, make sure the test suite finishes (and the engine stops) before `verify_matching.sh` starts — otherwise the GTC files written by the previous run will corrupt the initial book state.

10.7 The hardest problems: time, identity, and state

This section documents the traps that made the toolset non-trivial to build. Understanding them is useful when you extend the verification suite or adapt it to a different system.

10.7.1 Problem 1 — Order ids, now reproducible by design

`Order.create()` itself mints a fresh random UUID on every call, which would make `--seed 42` produce a different dataset on every run — fatal once amendments and cancellations need to name the order they act on. The generator works around this by minting its *own* deterministic id (`_order_id()`, seeded from the same `random.Random`) and overwriting `order.id` with it before the line is written; the FIX line then carries that id explicitly (`NEW|ID=<id>|...`), and both `replay_to_engine.py` and the live engine (`Order.from_dict`, reading straight off the wire) adopt it unchanged. So a `--seed 42` run really does produce exactly the same 1 056 order ids every time, in both the paper result and the engine result.

That still doesn't make `order_id` safe to compare directly, for an unrelated reason: the book snapshot used for comparison never exposes ids in the first place. **Solution (unchanged):** compare at price-level aggregate. The snapshot API returns `{price, qty}` pairs, not order ids — two different orders resting at $149.50 with total visible qty 450 are equivalent for book-state purposes regardless of what either side happened to call them.

10.7.2 Problem 2 — TIF=DAY orders expire at session end

`DAY` orders live only within the current session. The engine's scheduler marks them expired when the session transitions to `POST_TRADING` or `CLOSED`.

If the engine session ends between sending the last order and requesting the snapshot, all resting `DAY` orders vanish — and the book will appear empty.

Solution: the generated config sets `sessions_enabled: false`. That single line means no `pm-scheduler` is expected and the engine starts straight in `CONTINUOUS` — rather than the `true`-default behaviour of starting `CLOSED` and waiting for one to open the market — and simply stays there, since nothing ever schedules a transition out of it. All test orders use `TIF=DAY` (the default), which in a perpetual `CONTINUOUS` session never expires.

GTC orders add a different complication

`TIF=GTC` orders survive session transitions by being written to `gtc_orders.json`. The script's clean-state step deletes that file before each run, ensuring the engine starts from a blank slate. Without this step, GTC orders from a previous run would sit on the book and cause the MM seed orders to fill against unexpected resting liquidity.

10.7.3 Problem 3 — STOP prices depend on last_trade_price sequence

A STOP BUY at $150.50 only triggers if a trade prints at or above $150.50. Whether that trade happens at all depends on the exact order in which prior orders arrived. A single out-of-order message can shift `last_price` by one tick and prevent or trigger an entire cascade of stops.

Solution: The replay client uses a strict synchronous send-and-wait loop. No new line — `NEW`, `AMEND`, or `CANCEL` — is sent until the engine has confirmed (or timed out on) the previous one. This enforces the same processing sequence as the paper trader, which applies one dataset line at a time in a tight loop.

10.7.4 Problem 4 — Gateway IDs differ between paper and engine

The paper trader assigns `gateway_id="PAPER01"` to every order. The live engine is configured to accept only `gateway_id="VERIFY01"`.

At first glance this might seem to mean the two paths process different orders. In practice, `gateway_id` is stored on the `Order` object but plays no role in matching logic — it is used only for ACK routing and fill attribution. Two orders identical in symbol, side, type, price, and qty will produce identical book outcomes regardless of `gateway_id`.

10.7.5 Problem 5 — The drain window

The engine processes orders asynchronously via a ZeroMQ PULL socket. Even after the last ACK is received by the replay client, the engine may still be:

- Evaluating pending STOP triggers
- Writing fills back to the PUB socket

If a snapshot is requested before this work completes, the snapshot will be stale.

Solution: A 0.5 s pause (`DRAIN_PAUSE_S`) is inserted between the last ACK and the first snapshot request. This is a heuristic — it works reliably on modern hardware for 1 000 orders, but could theoretically fail on a very heavily loaded machine. The snapshot timeout is set to 5 000 ms as an additional safety net.

10.7.6 Problem 6 — Iceberg visible-qty refresh

An iceberg order has a `total_qty` and a `visible_qty`. When the visible slice is fully consumed, the engine replenishes it from the hidden reserve. The replenishment creates a *new* queue position for the refreshed slice.

In the paper trader, the reference replenishes the peak inline when it is used up. In the engine, `OrderBook.process()` does the same, but may interact with subsequent orders differently if message timing varies.

The comparison aggregates total visible qty at each price level, not per-order qty. As long as the total resting quantity at $419.03 is 450 in both cases, the iceberg's internal refresh state does not matter.

10.8 The verification config

`data/verify/verify_engine_config.yaml` is **not** a hand-maintained file — `gen_verification_set.py` writes a fresh copy of it (`build_engine_config()`) on every run, from the same `SYMBOLS`/`REF_PRICE` tables the FIX files are built from, so the two can never drift apart. It used to be a real file tracked outside the generator, gitignored under `data/**` and never actually committed — so a fresh checkout failed at the "start engine" step with nothing to deploy. Real output, verbatim:

```
# Generated by tools/gen_verification_set.py -- do not edit.
# Regenerated on every run, from the same SYMBOLS and REF_PRICE the
# orders are built from.

# The engine is the whole market here: no scheduler runs, so session
# handling has to be off or the engine sits in CLOSED and rejects
# every order with MARKET_CLOSED.
sessions_enabled: false

participants:
  - id: VERIFY01

symbols:
  AAPL:
  AMAZ:
  MSFT:
  GOOG:
```

Key decisions:

- Only `VERIFY01` is allowed — any existing production gateway that happens to be connected will be rejected, preventing order bleed.
- No `market_maker_orders` stanza — liquidity is injected via the FIX file itself, so the paper trader and engine see identical seeds.
- `sessions_enabled: false` — the engine starts straight in `CONTINUOUS` and stays there indefinitely (see Problem 2), avoiding `TIF=DAY` expiry.

10.9 Running the full suite

```
# One-shot end-to-end verification (takes ~10 s on a laptop)
./tools/verify_matching.sh

# Faster smoke test with fewer lines
./tools/verify_matching.sh --count 100

# Regression: fix the seed, run on every commit
./tools/verify_matching.sh --seed 42 --count 10000

# Reuse the existing FIX files to re-test after an engine code change
./tools/verify_matching.sh --skip-gen
```

Expected end-to-end output for a passing run (real, `--seed 42`, default `--count 10000` — elapsed time measured directly at ~10.6 s total, so the "~10 s on a laptop" estimate above still holds even though the default line count is ten times what it used to be):

```
──  STEP 1 — Generate verification dataset (seed=42, count=10000)  ──
[GEN] Generating and paper-trading 10000 line(s) …
[GEN] Wrote 56 MM orders  → data/verify/mm_orders.fix
[GEN] Wrote engine config    → data/verify/verify_engine_config.yaml
[GEN] Wrote 10000 test lines  → data/verify/test_orders.fix
      7106 new, 2380 amend, 570 cancel → 9020 trades
  AAPL    bids=56 levels  asks=65 levels  last=150.37
  AMAZ    bids=77 levels  asks=55 levels  last=179.92
  MSFT    bids=102 levels  asks=81 levels  last=419.92
  GOOG    bids=69 levels  asks=73 levels  last=160.21
[GEN] Saved paper result     → data/verify/paper_result.json
[GEN] Done — run replay_to_engine.py to get the engine result.

──  STEP 2 — Start matching engine  ──
[VERIFY] Engine data directory: /path/to/checkout/src/data
[VERIFY] Deploying verify_engine_config.yaml …
[VERIFY] Starting engine …
[VERIFY] Waiting for engine to bind (2 s) …
[VERIFY] Engine PID=84231 is running.

──  STEP 3 — Replay orders to engine  ──
[REPLAY] Gateway VERIFY01 authenticated.
[REPLAY] … 500 lines sent (500 acc, 0 rej)
...
[REPLAY] Sent 10056 lines in 5.94s  (10056 accepted, 0 rejected, 4 parse errors)
[REPLAY] Pausing 0.5s for engine to drain …
[REPLAY] Saved engine result → data/verify/engine_result.json

──  STEP 4 — Shut down engine  ──
[VERIFY] Sending SIGINT to engine …

──  STEP 5 — Compare paper vs engine  ──
  ✓  PASS  — 56 bid level(s), 65 ask level(s), last=150.37
  ✓  PASS  — 77 bid level(s), 55 ask level(s), last=179.92
  ✓  PASS  — 102 bid level(s), 81 ask level(s), last=419.92
  ✓  PASS  — 69 bid level(s), 73 ask level(s), last=160.21

══  RESULT: PASS  ══

✓  Verification PASSED — engine output matches paper trade.
```

10.10 Summary

Problem	Root cause	Solution
Order id comparison	Book snapshots don't expose ids at all	Compare at price-level aggregate (the generator's own ids *are* reproducible, for `AMEND`/`CANCEL`'s sake, but that's beside the point)
`TIF=DAY` expiry	Scheduler fires session transition	`sessions_enabled: false` → engine starts and stays `CONTINUOUS`
GTC state leak	Previous run writes to disk	Delete GTC/book-stats files in the engine's *real* data directory before each run
STOP trigger sequence	`last_price` depends on arrival order	Synchronous send-and-wait loop, for `NEW`, `AMEND`, and `CANCEL` alike

Problem	Root cause	Solution
Different `gateway_id`	Engine rejects PAPER01	Use `VERIFY01` in engine config; ID has no effect on matching
Async drain window	Engine still evaluating after last confirmation	0.5 s pause + 5 s snapshot timeout
Iceberg refresh position	Hidden reserve replenishment	Aggregate visible qty per price level

10.11 What next?

- Order Types — the full mechanics of each order type that the generator exercises
- Messages — the ZeroMQ message protocol used by the replay client
- Architecture — how the engine, gateway, and ZMQ bus fit together

CHAPTER 11

Message Generator (`pm-msgen`)

Learning objectives

After reading this page you will understand:

- Why a message described in three places drifts, and what a generator does about it that a code review does not
- How to read and write a message specification in `spec/messages/`
- How to use a generated binding: building, parsing, validating and subscribing without a single topic string literal
- Why `from_dict` deliberately accepts payloads that `validate()` rejects, and when you want each
- What `pm-msgen check` guarantees, and why generation must be byte-for-byte deterministic for that guarantee to be worth anything

What is covered

Sixteen families are specified and adopted — every family there is, covering 121 messages and 42 record types: `trade`, `order`, `session`, `book`, `log`, `index`, `risk`, `structure`, `quote`, `circuit_breaker`, `auction`, `drop_copy`, `admin`, `system`, `news` and `sim`. Each has a Python binding; `trade` and `order` also have C bindings for their CALF projections and the BALF `execution_report` frame.

All of it is live. `models/message.py`'s builders delegate to the generated code, the engine, pm-index and pm-log-srv publish through it, `md_gateway/normaliser.py` projects through it, and the CALF and BALF example clients parse with the generated structs. **No topic appears as a literal anywhere in `src/`** — including parameterised topics written as f-strings.

The drift check runs in CI and `make check`; compiled round-trip tests prove Python and C agree on both the text and binary wires.

`../../protocols-and-clients/part-5-message-reference/010-message-reference.md` is generated too. Its reference half — the topic index, the record types and one section per message — is rendered from the spec, and `pm-msgen check` fails when the page and the spec disagree. Its narrative half lives in `../../protocols-and-clients/part-5-message-reference/000-preamble.md`, which is hand-written and is where anything the spec cannot state belongs.

Every producer reaches the wire through its generated builder except two frames that opt out on purpose — the BALF `execution_report` and `index.history`'s legacy-archive replay — which `tests/test_msgen_adoption.py` pins.

11.1 The problem

A message in EduMatcher is described in at least three places, none of which is authoritative and none of which is checked against the others:

Surface	Where
The publisher's payload shape	a `make_*` factory in `models/message.py`, or an inline dict in the producer
The subscriber's expectations	a topic string literal, repeated across gateways and tools

Surface	Where
The documentation	../../protocols-and-clients/part-5-message-reference/010-mess hand-maintained

Nothing links them. Rename a field on the publisher side and the subscriber keeps compiling, keeps running, and simply stops seeing the value — no error, just wrong. That is the failure class this tool removes.

The fix is not more discipline. It is to make one file the source of truth and **generate** the rest, then have CI fail when the generated output and the spec disagree.

11.2 How it fits together

```
spec/transports.yaml          the transport registry (ZeroMQ patterns, config keys)
spec/messages/<family>.yaml   one file per message family — the source of truth
        |
        |  pm-msgen generate
        ▼
src/edumatcher/models/generated/<family>.py     committed, DO NOT EDIT
```

Generated files are **committed**. A reader browsing the repository never needs to run the generator, and `pm-msgen check` in CI proves the committed copy still matches the spec.

11.3 Commands

```
poetry run pm-msgen lint       # validate the spec only
poetry run pm-msgen generate   # write the generated files
poetry run pm-msgen check      # fail if committed output differs from the spec
```

```
poetry run pm-msgen grep-literals   # which topics are still hard-coded
```

Or through the Makefile, which is what `make check` and CI use:

```
make msgen          # regenerate — run this after editing a spec file
make msgen-check    # verify the committed bindings match spec/  [stamp-cached]
```

Exit codes follow the `pm-cverifier` convention: `0` success, `1` drift detected, `2` the spec itself is broken. The distinction matters for a build gate: `1` means "you forgot to regenerate", `2` means "your spec is wrong".

Both `generate` and `check` accept `--spec DIR` and `--out-python DIR`, which is mostly useful in tests. The defaults are relative to the repository root, so run them from there.

11.3.1 The normal edit loop

```
$ vim spec/messages/trade.yaml     # add a field, change a constraint
$ make msgen                       # regenerate
$ git add spec/ src/edumatcher/models/generated/
```

Forgetting the middle step is what `make check` and CI now catch:

```
$ make check
- Checking generated messages match the spec (pm-msgen)...
pm-msgen check: generated output is out of date with the spec.
Run `pm-msgen generate` and commit the result.

--- generated/trade.py (committed)
+++ generated/trade.py (from spec)
@@ -156,7 +156,7 @@
-        if self.tick_decimals > 8:
+        if self.tick_decimals > 6:
✗ Generated message bindings are out of date. Run 'make msgen' ...
```

11.4 Adding a family

1. Write `spec/messages/<family>.yaml`. Run `pm-msgen lint` until it passes.
2. Run `pm-msgen generate`. Review the generated file as you would any other.
3. Write a wire-compatibility test comparing the generated output against the existing hand-written producer for that family. **No family is adopted without one.**
4. Only then change call sites.

11.4.1 Two comparisons, not one

Step 3 needs care, because "byte-identical" is the right assertion in one direction and the wrong one in the other:

Comparison	Assert	Why
hand-written factory vs generated `make_*`	**byte-identical frames**	Both derive from a `to_dict()` over the same fields. There is no excuse for a difference.
an inline producer dict vs the generated payload	**equal key sets and equal values**	JSON objects are unordered and every consumer reads with `.get`, so key order is not part of the contract. Asserting byte-identity would be stronger than the system actually promises, and would block a legitimate change.

The `trade` family shows why the distinction matters: `engine/main.py::_publish_trade` emits `tick_decimals` between `price` and `quantity`, while `feed_schema.TradeExecutedPayload.to_dict()` emits it last. Both are correct. Nothing on the wire can tell. Only a byte-comparing test can, and it would be testing the wrong thing.

`tests/test_msgen_trade_wire_compat.py` makes both claims explicitly, and pins the divergence so it cannot be quietly forgotten.

11.5 Writing a specification

11.5.1 Family-level keys

A spec file has four top-level keys:

Key	Required	Meaning
`family`	yes	Must equal the filename stem: `heartbeat` → `spec/messages/heartbeat.yaml`. Also the topic root: `heartbeat.*`
`version`	yes	Integer layout version, exposed as `FAMILY_VERSION` in the generated module. Increment when the change is **not backward-compatible** — a field removed, renamed, or its type changed. Adding an optional field or tightening a `validate:` rule does not require a bump.
`types`	no	Named record type definitions shared across this family's messages. Referenced by `type: nested` and `type: list` fields.
`messages`	yes	Ordered list of message definitions.

11.5.2 A minimal family

```
family: heartbeat          # MUST equal the filename stem: spec/messages/heartbeat.yaml
version: 1

messages:
  - name: engine_alive
    topic: "system.heartbeat"
    transport: [engine_pub]
    doc:
      motivation: "Liveness ping so a subscriber can tell a quiet engine from a dead one."
    fields:
      - name: sent_at
        type: float
        required: true
        unit: epoch_seconds
        doc: When the engine emitted this ping.
```

That is enough to generate a dataclass, a validating constructor, a parser and a topic constant.

11.5.3 Message-level keys

Key	Required	Meaning
`name`	yes	snake_case identifier. Becomes the suffix of every generated symbol: `make_<name>`, `parse_<name>`, `TOPIC_<NAME>`
`topic`	yes*	Topic string, e.g. `"order.ack.{gateway_id}"`. May contain `{field_name}` parameters. Omit only for binary-only messages (`transport: [balf]`) that never touch the bus.
`transport`	yes	List of transports this message uses. Names must exist in `spec/transports.yaml`.
`doc`	yes	Documentation block — see sub-keys below.
`fields`	yes	Ordered list of field definitions — see Field keys.
`encoding`	no	Per-transport binary or text projection details (`balf:`, `calf:`, `ralf:`). Required when a transport carries a non-JSON wire format.

The `doc:` block sub-keys:

Sub-key	Required	Meaning
`motivation`	yes	One or two sentences on why this message exists. Appears in the generated reference.

Sub-key	Required	Meaning
`published_by`	yes	List of process roles from the closed vocabulary: `engine`, `gateway`, `scheduler`, `index`, `stats`, `clearing`, `md_gateway`, `api_gateway`, `admin`, `log_server`, `log_client`
`since`	no	Version string when the message was introduced, e.g. `"1.0"`.
`see_also`	no	List of related message names for generated cross-references.
`example_note`	no	Prose about edge cases or usage subtleties shown in the generated reference.

11.5.4 Field keys

Key	Meaning
`name`	snake_case identifier, unique within the message
`type`	`string`, `int`, `float`, `bool`, `enum`, `ticks`, `nested`, `list`. `ticks` is an integer price in engine ticks — it generates the same `int` as `type: int` but makes the unit self-documenting and lets binary layout entries distinguish tick prices from plain integers
`ref`	for `nested`, and for a `list` **of records**: the name of a family-level entry under `types:`
`item`	for a `list` **of scalars**: the element type. Exactly one of `ref` / `item`
`required`	default `true`. A `required: false` field **must** say which presence regime it means — see below
`default`	what a *producer* gets when it omits the field. Must be a legal value
`nullable`	the value may be `None`; the key is still always emitted, as `null`
`omit_when_none`	implies `nullable`, and **omits the key entirely** when the value is `None`
`omit_when_empty`	strings and lists. Omits the key when the value is `""` / `[]`
`parse_default`	what `from_dict` substitutes when the key is missing from an *inbound* payload. Need not be legal — see Coercion vs validation
`unit`	required on every numeric field. One of `display_price`, `ticks`, `shares`, `epoch_seconds`, `epoch_nanos`, `duration_nanos`, `percent`, `dimensionless`, `money`
`doc`	prose for the generated documentation and the `describe_*()` table
`values`	required for `type: enum`; declaration order is authoritative
`validate`	`gt`, `ge`, `lt`, `le`, `max_len`, `min_len`, `min_items`, `max_items`, `pattern`

The loader is strict, on purpose

An unknown key **raises**. It is not ignored, and it is not warned about.

```
SpecError: spec/messages/order.yaml: messages[0] ('order_ack').fields[3]:
  unknown key(s) 'requird' (did you mean 'required'?)
```

A silently-ignored `requird: true` would disable a field with no error, which is precisely the class of bug the generator exists to kill. Tolerating it here would defeat the whole tool.

11.5.5 Presence: four regimes, stated explicitly

An optional field is one of four things on the wire, and the loader will not guess which:

```
    # 1. always present, "" when the producer omits it
    - { name: reason, type: string, required: false, default: "" }

    # 2. always present, null when unset
    - { name: price, type: float, required: false, nullable: true,
        unit: display_price }

    # 3. absent entirely when unset
    - { name: client_tag, type: string, required: false, nullable: true,
        omit_when_none: true, validate: { max_len: 64 } }

    # 4. absent when the value is empty (strings and lists only)
    - { name: prev_state, type: string, required: false,
        omit_when_empty: true, validate: { max_len: 32 } }
```

Regime 4 keys on `""` rather than on `None`, and it is the one the codebase used most before any of this existed: 27 hand-written builders drop a key with `if x:`. It is deliberately narrow — **strings and lists only**, since on a number it would silently drop a legitimate zero and `""` is not a declared value of an enum — and it cannot be combined with either `omit_when_none` (a field omits on one or the other, not both) or `default` (the empty value *is* the absence here, so there is nothing for a default to supply). Both combinations are loader errors.

On a list it is exactly symmetric with the read: an optional list already reads back through `p.get(key, [])`, so absent and `[]` are the same value to every reader, and omitting one is the same statement as emitting the other. That makes it the regime for a field that only some variants of a record carry — `index`'s `HistoryRecord` is five archived shapes in one record, and without it four of them would grow an `"constituents": []` they never had. A `min_items` above zero contradicts it and is a loader error: a list that must carry an item can never be empty, so the omission could never fire.

```
>>> _topic, payload = decode(make_cancelled_msg("GW1", "O1"))
>>> payload
{'order_id': 'O1'}                      # no client_tag key at all
```

Why omission is modelled rather than replaced by null

`models/message.py` states the reason: *"an ordinary single order carries none of these, and its events should not grow four empty fields to say so."* Emitting `null` would have been simpler grammar and a worse wire.

One flag is enough because **absence and `null` are indistinguishable to every reader here** — verified, not assumed: the engine's amend handler does `payload.get("price")` and tests `is None`, never `in payload`. A protocol where presence itself carried meaning would need a third state, and this one does not.

`validate()` rules apply only when a nullable field is set — `None` means "not set", not "set to something invalid", so a `max_len` on an absent `client_tag` does not fire.

11.5.6 Why `unit` is mandatory on numbers

`unit` never converts anything. It is documentation that the reviewer can see and the generator can print. It exists because `trade.executed.price` is display money while `trade_log.price` is ticks — a mismatch that cost a full session to untangle, and which no type system catches because both are numbers. Declaring it makes it reviewable.

```
    - name: price
      type: float
      unit: display_price      # not ticks; the publisher already converted
      validate: { gt: 0 }
```

11.5.7 Records: `types:`, `nested` and `list`

A family may declare record types once, at the top of its file, and reference them by name. This is how `order.oco` carries two legs and how a book snapshot carries two price ladders and a trade tape.

```
family: book
version: 1

types:
  BookLevel:
    doc: "One aggregated price level."
    fields:
      - { name: price, type: float, unit: display_price }
      - { name: qty,   type: int,   unit: shares }
      - { name: count, type: int,   unit: dimensionless }

messages:
  - name: book_snapshot
    topic: "book.{symbol}"
    transport: [engine_pub]
    doc:
      motivation: "Aggregated view of one instrument's order book."
    fields:
      - { name: symbol, type: string, validate: { max_len: 16 } }
      - { name: bids, type: list, ref: BookLevel }
      - { name: asks, type: list, ref: BookLevel }
```

The generator emits one `@dataclass` per type, **before** the messages that embed it, with the same `from_dict` / `to_dict` / `validate` trio a message gets. Rules declared on a record's fields therefore apply everywhere it is embedded, which is the point of declaring it once:

```
>>> snap = BookSnapshot.from_dict(payload)
>>> snap.bids[0].price          # a real BookLevel, not a dict
95.0
>>> snap.validate()             # walks every level of both ladders
```

A `nested` field holds one record; combine it with `nullable` + `omit_when_none` to say **both-or-neither** for a group of fields that only make sense together:

```
      - { name: next, type: nested, ref: NextTransition, required: false,
          nullable: true, omit_when_none: true }
```

That is why the IDL has no `co_present: [a, b]` constraint. A pair of keys that must travel together is a record that was flattened into `a_b` names, and a nullable record makes the half-set state *unrepresentable* rather than merely invalid. `session.state` is the worked example: `next_state` and `next_at` only make sense together, so they live in a nullable `NextTransition` record instead of two co-present flat keys.

A `list` takes `min_items` / `max_items`, which are ordinary `validate:` rules enforced in both bindings:

```
      - { name: legs, type: list, ref: ComboLeg,
          validate: { min_items: 2, max_items: 10 } }
```

A list of **scalars** uses `item:` instead of `ref:`:

```
      - { name: processes, type: list, item: string, required: false }
```

Absent and empty are the same thing to a list, so an optional one reads back as `[]` and needs no `default:` — declaring a non-empty one is an error, since it would put a value nobody chose on the wire. Scalar rules (`max_len`, `gt`, `pattern`, …) are rejected on a list: they would silently do nothing. A scalar list keeps its `make_*_unchecked`, since it embeds no record.

What records deliberately do not do

Each of these is an error in a spec file rather than a wrong answer in a committed binding:

Rejected	Why
a **cycle** in the type graph (`A` embeds `B` embeds `A`)	no finite payload could satisfy it. Depth is fine: a record may embed another to any depth, and types are emitted in dependency order
a record or list on `calf` / `balf`	a record inside a key-value line or a fixed binary frame is an unsolved layout question
a `nullable` list	an empty list already says "nothing"; null would be a second spelling every reader must handle
a type nothing references	it would generate a class nothing constructs
`min_items` above `max_items`, or a negative bound	no list could satisfy it, so *every* message would fail `validate()` at runtime

A message containing a record or a list also gets **no `make_*_unchecked`**. That builder is a dict literal and a record has no literal form; neither combo, OCO nor a book snapshot is a measured hot path, so omitting it is more honest than emitting a slow function under a name that promises speed. The omission is per message — a flat message in the same family still has one.

Maps are not supported, deliberately

A spec that appears to need one is usually describing a message that should have been a list of records. `leg_fill_qty: {0: 5}` only ever meant `legs[0].filled_qty = 5`, and the list index was already the key.

11.5.8 Topics with parameters

A topic segment may be a `{placeholder}` naming a field of the message:

```
  - name: order_ack
    topic: "order.ack.{gateway_id}"
    transport: [engine_pub]
    doc:
      motivation: "Acknowledge acceptance or rejection of a new order."
    fields:
      - { name: gateway_id, type: string, validate: { max_len: 32 } }
      - { name: order_id,   type: string, validate: { max_len: 64 } }
      - name: status
        type: enum
        values: [ACCEPTED, REJECTED]
```

This is what removes scattered topic literals from subscribers. See Example 4.

11.5.9 Transports

A message names transports from `spec/transports.yaml` rather than restating a pattern and address per message:

```
transports:
  engine_pub:
    pattern: PUB
    subscriber_pattern: SUB
    address_config_key: ENGINE_PUB_ADDR    # symbolic; resolved at runtime
```

`address_config_key` must be a symbolic name, never a literal address — that is what keeps ports out of generated code. `pm-msgen lint` rejects anything that looks like `tcp://...`.

All three external protocols are generated: `calf` and `ralf` (key-value text lines) and `balf` (fixed binary frames).

11.6 Binary messages

A BALF message declares a byte layout instead of wire keys:

```yaml
- name: execution_report
  transport: [balf]          # no `topic:` — it never touches the bus
  fields:
    - { name: order_id, type: int, unit: dimensionless }
    - { name: fill_price, type: float, unit: display_price, validate: { gt: 0 } }
    - { name: symbol, type: string, validate: { max_len: 8 } }
    - name: side
      type: enum
      values: [BUY, SELL]
  encoding:
    balf:
      msg_type: 0x20
      frame_size: 64          # 8-byte header + 56-byte body
      price_scale: 100000000
      layout:
        - { field: order_id,   repr: u64,       offset: 8 }
        - { field: fill_price, repr: i64,       offset: 16, scale: price_scale }
        - { field: symbol,     repr: "char[8]", offset: 40 }
        - { field: side,       repr: u8,        offset: 48, enum_map: { BUY: 1, SELL: 2 } }
        - { reserved: 6, offset: 50 }
```

Note `repr: "char[8]"` is **quoted**: unquoted, YAML reads `[` as the start of a flow sequence.

11.6.1 Rules the loader enforces

Rule	Why
Every body byte is covered exactly once	A gap must be an explicit `reserved` run, so a hole is a decision rather than an oversight
Offsets may not overlap or overrun `frame_size - 8`	Two fields sharing a byte is undecodable
Every field needs a layout entry	A field with nowhere to go would silently never be sent
`char[N]` must equal the field's `max_len`	Otherwise a legal value would not fit the frame the spec describes
A binary enum needs a complete `enum_map`	A missing name is a value that cannot be encoded — discovered at runtime otherwise
`msg_type` is unique per transport across all families	It is all a receiver has to tell frames apart
A `repr` must be able to carry the field's type	`u64` cannot hold a string

The coverage rule is the one that earns its keep

A customer reference parser once modelled `order_id` as `char[16]` (a 16-byte string) instead of the correct `u64`, making it exactly 8 bytes too large on every message that carries one. Six message types were wrong. Nothing caught it because the example tested its parser only against frames the same file built — it agreed with itself while disagreeing with the gateway. The coverage rule would have rejected the spec at load time: `body byte(s) 48-55 are not covered by any layout entry`.

11.6.2 Using it

```
from edumatcher.models.generated.order import (
    serialise_execution_report_balf,
    parse_execution_report_balf,
    FRAME_SIZE_EXECUTION_REPORT_BALF,   # 64
)

frame = serialise_execution_report_balf(payload, seq_no=session.next_seq())
sock.sendall(frame)                      # exactly 64 bytes

report = parse_execution_report_balf(frame)
report.fill_price     # 150.0 — the i64/1e8 scaling is undone for you
report.side           # "BUY"
```

The **8-byte header is the generator's**, not the spec's: magic `0xBA`, version, `msg_type`, flags and a `u32` sequence number. It must not appear in `layout`, and `parse_*` validates all of it plus the frame length before reading a single field.

Why length is checked before anything else

BALF frames are fixed-size per `msg_type`. A frame of the wrong length is *not this message*, and unpacking it anyway would read neighbouring bytes as field values and hand back a plausible-looking result. That is the "no error, just wrong" failure this generator exists to remove, so both bindings refuse rather than guess.
In C the same message is a struct whose types say what the values *are*, not what carries them — a scaled `i64` price becomes a `double`, an enum byte becomes an enum:

```
edu_execution_report_balf_t er;
int rc = edu_execution_report_balf_parse(frame, len, &er);
if (rc != EDU_MSG_OK) { fprintf(stderr, "%s\n", edu_msg_strerror(rc)); return; }

er.fill_price;   /* double, already divided by 1e8 */
er.order_id;     /* uint64_t */
edu_execution_report_side_to_str(er.side);   /* "BUY" */
```

Values are read byte-wise rather than through a cast, because BALF bodies are packed rather than aligned — `side` sits at body offset 48 with no padding — and a misaligned cast is undefined behaviour.

11.7 Projections: one event, several shapes

The three client-facing encodings of a trade are **not the same fields under three names**. They are different *projections* of the engine's payload — a subset of fields, each renamed, some supplied by the gateway rather than the message:

Field (bus `trade.executed`)	bus JSON	CALF `TRADE`	RALF `EXEC`
`id`	`id`	— not on the public feed	`EXEC_ID`, `MATCH_ID`
`symbol`	`symbol`	`SYM` (gateway)	`SYM` (gateway)
`buy_order_id`	`buy_order_id`	—	`BUY_ORDER_ID`
`price`	`price`	`PX`	`PX`
`quantity`	`quantity`	`QTY`	`QTY`
`aggressor_side`	`aggressor_side`	`SIDE`	`SIDE`
`ts_ns`	`ts_ns`	`TS` (gateway)	`TS` (gateway)
`tick_decimals`	`tick_decimals`	—	—

A transport is declared with three keys:

```
    calf:
      msg_type: TRADE
      include: [price, quantity, aggressor_side]   # what this feed carries
      keys: { price: PX, quantity: QTY, aggressor_side: SIDE }
      gateway_injected: [CH, SYM, SEQ, TS]         # documentation only
```

`keys` may map one field to several wire names (RALF's `id: [EXEC_ID, MATCH_ID]`).

A topic parameter is dropped from the bus payload by default

`order.ack.{gateway_id}` names the gateway in its topic and does **not** repeat it in the body, which is what the hand-written builder did — so that is the default.

`book.{symbol}` is the other case: `OrderBook.snapshot()` puts `symbol` in the body *as well*, and every subscriber reads it from there. Say so with an explicit `include:` that lists the parameter:

```
    engine_pub:
      frames: [topic, json_payload]
      include: [symbol, tick_decimals, bids, asks, ...]
```

"Named in the topic" and "absent from the body" are two different facts, and only the second is a property of the wire. The first generated `book.{symbol}` binding silently lost the key because the default conflated them.

The generated Python side is a pair of functions per transport:

```
from edumatcher.models.generated.trade import (
    project_trade_executed_calf,   # bus payload  -> {PX, QTY, SIDE}
    parse_trade_executed_calf,     # {PX, QTY, SIDE} -> TradeExecuted
)

project_trade_executed_calf(payload)
# {'PX': '101.5', 'QTY': '300', 'SIDE': 'BUY'}
```

`project_*` takes a **payload mapping and reads only the fields this transport carries**. That matters more than it looks: a CALF gateway holds a trade with three relevant fields, and requiring it to supply `id`, `buy_order_id` and the rest — which CALF drops — would re-couple exactly the surfaces the projection model separates. A projection is a subset, so it depends on a subset.

```
# Everything CALF drops may be absent:
project_trade_executed_calf({
    "symbol": "AAPL", "price": 151.5, "quantity": 25, "aggressor_side": "BUY",
})
# {'PX': '151.5', 'QTY': '25', 'SIDE': 'BUY'}
```

Values are coerced to their declared types first, so `project_*(payload)` and `project_*(msg.to_dict())` always agree.

gateway_injected is documentation, not behaviour

The generated projection emits **only the included payload fields**. It never emits `CH`/`SYM`/`SEQ`/`TS`, and their order in the spec means nothing.

This is not an arbitrary choice. `md_gateway`'s `_emit_stream_event` puts `{CH, SYM, SEQ, TS}` *before* the payload; `ralf_gateway`'s `_emit_event` appends `SEQ` *after* it. No single "injected keys go here" rule can describe both. The envelope belongs to the gateway, the payload map to the generator.

11.8 Generated C

For every declared text projection the generator emits a typed struct and a parser into `docs/examples/generated/`:

```
docs/examples/generated/
    edumatcher_msg.h/.c        hand-written: error codes + edu_msg_strerror
    edumatcher_trade.h/.c      generated from spec/messages/trade.yaml
```

`edumatcher_msg.h` is the C counterpart of `_runtime.py` — the one file there that is *not* generated.

11.8.1 What a projection looks like in C

```
typedef enum {
    EDU_TRADE_EXECUTED_AGGRESSOR_SIDE_BUY = 1,
    EDU_TRADE_EXECUTED_AGGRESSOR_SIDE_SELL = 2,
    EDU_TRADE_EXECUTED_AGGRESSOR_SIDE_AUCTION = 3
} edu_trade_executed_aggressor_side_t;

typedef struct {
    double   price;                                    /* PX,  display_price */
    int64_t  quantity;                                 /* QTY, shares        */
    edu_trade_executed_aggressor_side_t aggressor_side;   /* SIDE               */
} edu_trade_executed_calf_t;

int edu_trade_executed_calf_parse(const calf_message_t *in,
                                  edu_trade_executed_calf_t *out);
int edu_trade_executed_calf_validate(const edu_trade_executed_calf_t *m,
                                     char *err, size_t errlen);
const char *edu_trade_executed_aggressor_side_to_str(
    edu_trade_executed_aggressor_side_t v);
```

Three fields, not eleven: **a C struct mirrors what its transport carries**, not the internal bus payload. C clients speak CALF and never see the bus.

Fixed-size buffers, no allocation, `int` returns — matching the hand-written example clients, so generated code drops in beside them. A `string` field becomes `char name[max_len + 1]`, which is why `validate.max_len` is mandatory for any string reaching an external transport.

11.8.2 Example 7 — reading a trade in C

```
#include "edumatcher_trade.h"

calf_message_t msg;
if (calf_parse_line(line, &msg) != 0) return;          /* hand-written tokeniser */

edu_trade_executed_calf_t trade;
char err[128];

int rc = edu_trade_executed_calf_parse(&msg, &trade);
if (rc != EDU_MSG_OK) {
    fprintf(stderr, "TRADE: %s\n", edu_msg_strerror(rc));
    return;
}
if (edu_trade_executed_calf_validate(&trade, err, sizeof(err)) != EDU_MSG_OK) {
    fprintf(stderr, "TRADE rejected: %s\n", err);       /* e.g. "price must be > 0" */
    return;
}

printf("%lld @ %g (%s)\n",
       (long long)trade.quantity, trade.price,
       edu_trade_executed_aggressor_side_to_str(trade.aggressor_side));
```

Note the **two steps**, mirroring `from_dict` and `validate()` in Python: `parse` coerces and reports a missing or unparseable field; `validate` enforces the declared rules. A client that would rather display a questionable print than drop it simply skips the second call. That is the same coercion/validation split described above, in a second language.

11.8.3 Building against the generated headers

```
GEN_DIR := ../generated
CFLAGS  := -std=c11 -Wall -Wextra -I. -I$(GEN_DIR)

SRC := your_client.c calf_parser.c \
       $(GEN_DIR)/edumatcher_trade.c $(GEN_DIR)/edumatcher_msg.c
```

`-I.` is needed as well as `-I$(GEN_DIR)`: the generated header includes `"calf_parser.h"`, which lives with the CALF example rather than beside it.

11.8.4 Error codes

Code	Constant	Meaning
`0`	`EDU_MSG_OK`	success
`-1`	`EDU_MSG_ERR_SHORT`	frame or line too short
`-2`	`EDU_MSG_ERR_MAGIC`	bad magic byte (binary only)
`-3`	`EDU_MSG_ERR_VERSION`	unsupported version (binary only)
`-4`	`EDU_MSG_ERR_MSGTYPE`	unknown or unexpected `msg_type`
`-5`	`EDU_MSG_ERR_LENGTH`	length mismatch (binary only)
`-6`	`EDU_MSG_ERR_FIELD`	field missing, unparseable, or rule failed
`-7`	`EDU_MSG_ERR_OVERFLOW`	value exceeds a fixed-size buffer

`-1..-5` mirror `balf_parser.c`'s `parse_header`/`split_frame`, whose logic the generated BALF parser reimplements.

A return code is a per-function contract, not a global registry

`calf_parser.c`'s hand-written `calf_parse_line` **also** returns `-1..-6`, with entirely different meanings — its `-4` is "too many fields", not "unknown msg_type", and its `-6` is "empty field key".
So check each call's result against the function you called, and use `edu_msg_strerror` only for functions declared in a generated header. There is no single return-code convention shared across the tree to reuse.

11.9 How the two bindings are kept honest

`tests/test_msgen_calf_roundtrip.py` compiles the **committed generated C** — the same files `docs/examples/calf` links against — with `-Wall -Wextra -pedantic -Werror`, feeds it lines built by the **committed generated Python** projection, and compares field by field. It also asserts that C and Python reject the same values.

This is the property no test in this repository could previously state: not "the generator works", but "a C client and a Python publisher read the same bytes the same way".

It follows `test_alf_examples.py` — `shutil.which("cc")` plus `pytest.skip`, no new dependency and no marker. `cffi` is not a dependency, the skip pattern makes a marker redundant, and `ubuntu-latest` ships a compiler.

11.10 Using a generated binding

Everything below uses the committed `trade` family. Import from `edumatcher.models.generated.trade`.

11.10.1 What you get

Symbol	Purpose
`TradeExecuted`	frozen dataclass, one attribute per field
`TOPIC_TRADE_EXECUTED`	the topic constant
`is_trade_executed(topic)`	topic test, for messages with no parameters
`topic_*()` / `PREFIX_*` / `match_*()`	build, subscribe and destructure a *parameterised* topic
`make_trade_executed(**kw)`	coerce, validate, return the two bus frames
`make_trade_executed_unchecked(**kw)`	identical frames, no validation — hot paths only. **Not emitted** for a message carrying a `nested` or `list` field

Symbol	Purpose
`parse_trade_executed(frames)`	frames to a validated object
`.from_dict()` / `.to_dict()`	interop with existing dict-based code
`.validate()`	standalone strictness check
`describe_trade_executed()`	field metadata at runtime, for spy tools
`FAMILY`, `FAMILY_VERSION`, `FAMILY_TOPICS`	registry, for routers

11.10.2 Example 1 — publishing

```
from edumatcher.models.generated.trade import make_trade_executed

frames = make_trade_executed(
    id="42",
    symbol="ACME",
    buy_order_id="b-1",
    sell_order_id="s-1",
    buy_gateway_id="GW1",
    sell_gateway_id="GW2",
    price=101.5,
    quantity=300,
    aggressor_side="BUY",
    ts_ns=1_700_000_000_000_000_000,
    tick_decimals=2,
)
# [b'trade.executed',
#  b'{"id":"42","symbol":"ACME",...,"tick_decimals":2}']

publisher.send_multipart(frames)
```

`make_*` returns **exactly two frames**: topic and payload. Everything behind them is added at publish time by the wrappers in `messaging/bus.py`, never by `make_*` — adding either here would double-stamp every message:

Frame	Added by	Contains
0–1	`make_*`	topic, JSON payload
2	`SequencedPublisher`	per-topic sequence number
3	`CausalPublisher`	causal envelope (`msg_id`, `causation_id`, `correlation_id`)

On the PUSH side there is no sequence, so `CausalPusher` puts the envelope at frame 2 instead. Readers find it by shape rather than by index — see `models/message.py::decode_envelope`.

Missing and mistyped arguments

`make_*` takes keyword arguments and routes them through `from_dict`, so it coerces on the way in. `price=100` (an `int`) puts `100.0` on the wire, the same as the hand-written factory does. A missing required field raises `KeyError`, matching the existing payload classes.

11.10.3 Example 2 — consuming

```
from edumatcher.models.generated.trade import (
    is_trade_executed,
    parse_trade_executed,
)
from edumatcher.models.message import decode

frames = subscriber.recv_multipart()
topic, _payload = decode(frames)

if is_trade_executed(topic):
    trade = parse_trade_executed(frames)
    print(f"{trade.symbol} {trade.quantity} @ {trade.price} ({trade.aggressor_side})")
```

`parse_*` coerces *and* validates, so a malformed payload raises `MessageValidationError` at the boundary rather than producing a plausible- looking object that fails somewhere deeper. It reads only the first two frames, so a message carrying a sequence and a causal envelope parses unchanged — which is why neither addition needed a single consumer to change.

11.10.4 Example 3 — validating without parsing

You often have a payload dict already (from a log, a replay file, a database row). Skip the frames:

```
from edumatcher.models.generated.trade import TradeExecuted
from edumatcher.models.generated._runtime import MessageValidationError

trade = TradeExecuted.from_dict(row)     # coerces, never raises on a rule
try:
    trade.validate()
except MessageValidationError as exc:
    log.warning("archived trade %s is not spec-conformant: %s", row.get("id"), exc)
```

This split is the whole point of the next section.

11.10.5 Example 4 — subscribing without a topic literal

For a parameterised topic the generator emits three helpers, which is what removes hand-typed topic strings from subscribers:

```
from edumatcher.models.generated.orders import (   # illustrative
    PREFIX_ORDER_ACK,
    match_order_ack,
    topic_order_ack,
)

sock.setsockopt_string(zmq.SUBSCRIBE, PREFIX_ORDER_ACK)   # "order.ack."

topic, _payload = decode(sock.recv_multipart())
gateway_id = match_order_ack(topic)      # "GW1", or None if it isn't this topic
if gateway_id is not None:
    ...

# and to publish:
sock.send_multipart(encode(topic_order_ack("GW1"), payload))
```

Why a single segment, not a greedy match

`match_*` matches a single dot-delimited segment. A greedy `.+` would make `order.ack.GW1.extra` match and return `"GW1.extra"` — a subtly wrong gateway id rather than a clean `None`. The generated regex is `^order\.ack\.(?P<gateway_id>[^.]+)$`.

11.10.6 Example 5 — runtime field metadata

`describe_*()` returns the spec's own field table, which is what lets a spy or pretty-printer render units and constraints without hard-coding them:

```
from edumatcher.models.generated.trade import describe_trade_executed

for field in describe_trade_executed():
    unit = f" [{field['unit']}]" if field["unit"] else ""
    print(f"{field['name']:<16}{field['type']:<8}{unit}")

# id              string
# symbol          string
# ...
# price           float   [display_price]
# quantity        int     [shares]
# aggressor_side  enum
# ts_ns           int     [epoch_nanos]
# tick_decimals   int     [dimensionless]
```

11.10.7 Example 6 — the hot path

`make_*` validates on every call. The engine publishes per match, and the perf notes are explicit about microseconds, so the generator also emits a non-validating twin. This is what `engine/main.py::_publish_trade` actually does today:

```
from edumatcher.models.generated.trade import make_trade_executed_unchecked

self.pub_sock.send_multipart(make_trade_executed_unchecked(**trade.to_wire()))
```

`Trade.to_wire()` is where the model-to-wire conversion lives — `price` is **ticks** on the model and **display money** on the wire, and the scale comes from the tick registry keyed on the symbol, never from `Trade.tick_decimals` (which `Trade.create` defaults to 2 and the engine never sets, so it is wrong for any instrument that does not trade in hundredths). Expanding those kwargs at the publish site, as this snippet used to, is exactly how that conversion gets forgotten.

Note the signature difference: `make_*` takes `**kw` because its callers have a dict of uncertain provenance, while `make_*_unchecked` takes **explicit keyword-only typed parameters**, so a typo is a `TypeError` at the call site rather than a missing key on the wire.

The two produce **byte-identical frames** for any input — not just valid input — which is the only thing that makes the unchecked variant safe to reach for. See the cost of the hot path for why it is built the way it is.

Use _unchecked only on a measured path

It exists for the engine's trade publication loop, not for convenience. Everywhere else, the validation is the point: it is what stops a malformed message reaching a subscriber that cannot tell.

11.11 The cost of the hot path

Adopting a generated constructor on a per-match path is the one place this tool can make the system *worse*, so the decision was measured rather than assumed. 200 000 iterations, `orjson`:

Construction	µs/call	vs. hand-written	Byte-identical to `make_*`?
the hand-written dict literal it replaced	0.96	—	n/a
generated dict literal, **no** coercion	1.12	+0.16	**No**
generated dict literal, inline coercion — what ships	**1.47**	**+0.50**	Yes
via `from_dict` → dataclass → `to_dict`	4.03	+3.08	Yes

Two things fell out of that table, both of which changed the implementation:

The obvious implementation was unusable. Building `_unchecked` on top of `from_dict` and `to_dict` is the natural way to guarantee the two constructors agree, and it costs +3.1 µs. `perf-notes.md` records publication optimisations worth 0.2–1.0 µs *each*; this would have undone all of them several times over. A function whose entire purpose is "measured hot paths only" cannot be 4× slower than the code it replaces. So `make_*_unchecked` is generated as a direct dict literal with the topic pre-encoded at import — the same optimisation the engine's own `_TRADE_TOPIC` constant was, now generated instead of hand-written.

Dropping coercion was tempting and wrong. It saves a further 0.34 µs, and the engine already passes correctly-typed values, so it looks free. It is not: `make_*_unchecked(price=100)` would then put `100` on the wire where `make_*` puts `100.0`. And mypy does **not** catch it — `int` is promotable to `float`, so `price=100` against `price: float` type-checks clean. That is a silent divergence between two functions documented as producing identical frames, which is the exact failure class in The problem. The 0.34 µs is paid.

`tests/test_msgen_trade_perf.py` (marker `perf`, deselected by default) guards against a reversion to the 4× shape. Its threshold is deliberately loose — 3×, against a measured 1.5× — because it is a guard, not a benchmark, and CI timing is noisy.

Could the generated code be as fast as the hand-written literal?

No, and the reason is structural rather than a shortcoming of the emitter. The short version:

- **~53 % of the original call was already `orjson`** — 0.51 µs of 0.96. That half is untouched either way.
- **+0.165 µs is the cost of having a function at all**: a frame push and eleven keyword bindings, versus a literal the compiler inlines at the call site. A shared definition is a call; a copied one is not. This is the floor for any generated builder.
- **+0.393 µs is coercion**, and it is *call overhead* (~36 ns × 11), not conversion work — `str()` on a `str` returns the argument unchanged.

One real optimisation remains: coercing only the numeric fields halves the overhead to +0.27 µs, because a type checker already rejects `int`-where-`str` and `float`-where-`int`; it cannot reject `int`-where-`float` or `bool`-where-`int`. It is not applied because it weakens `_unchecked`'s "byte-identical for any input" promise to "for input a type checker has seen" — `make_*_unchecked(**payload_dict)` defeats it.

Also measured and worth knowing: replacing the coercion call with a `price.__class__ is float` test is **slower**, not faster.

11.12 Adopting a binding in existing code

Wiring a generated binding into a producer or consumer that predates it comes in a few recognisable shapes.

11.12.1 A delegating shim

A hand-written `make_*` factory becomes a one-line delegation:

```
def make_trade_msg(trade_dict: dict[str, Any]) -> list[bytes]:
    return _gen_trade.make_trade_executed(**trade_dict)
```

Byte-identical output, with **one deliberate behaviour change: it now validates.** A zero price, or a payload with no `aggressor_side`, previously went out on the wire without complaint and now raises `MessageValidationError`. That is the point — producers are held to the contract — and because the error subclasses `ValueError`, callers already guarding with `except ValueError` keep working.

This is the change most likely to surprise you

If something that used to publish now raises, it was publishing something the spec says is invalid. Fix the producer, or — if the spec is wrong — fix the spec. Do not reach for `make_*_unchecked` to make the error go away; that variant is for measured hot paths, not for silencing a real finding.

11.12.2 The producer itself

Replacing an inline dict literal with a generated builder removes the field list from the producer — it now lives only in the spec. That is where the value is: adding a field to a message becomes one edit to its `.yaml`, and the C clients pick it up too.

Published **key order may change** as a side effect. No consumer can observe this — JSON objects are unordered and every reader uses `.get` — which is why a wire-compatibility test for a producer asserts equal keys and values rather than equal bytes.

11.12.3 A recorder: topic constant, not the parser

Adopt the **topic constant** everywhere — it is pure gain and zero risk: a topic rename in the spec then reaches the subscriber instead of silently leaving it subscribed to a topic nobody publishes.

Do **not** reflexively adopt `parse_*`. A statistics recorder or archive replayer is often deliberately tolerant — skipping a partial print, falling back to receipt time for a missing timestamp, accepting a non-numeric id. `parse_*` validates, so adopting it there would make the recorder **raise** on inputs it handles on purpose.

A recorder records what it received

Refusing to store a message because it fails the current spec destroys exactly the evidence you need to find out why it was malformed. Adopt `parse_*` only where the consumer genuinely wants to reject a non-conforming message rather than record it. Do not assume every subscriber wants validation.

11.13 Coercion and validation are different jobs

This is the one rule worth internalising, because it is unusual and it is deliberate.

Function	Coerces?	Validates?
`from_dict(payload)`	**yes**	**no**
`validate()`	no	**yes** — the only strictness gate
`make_*(**kw)`	yes	yes
`make_*_unchecked(**kw)`	yes	no
`parse_*(frames)`	yes	yes
`to_dict()`	no	no

Why: the spec should state the honest contract, but the system has an archive written before that contract existed. Those two requirements conflict unless reading and asserting are separated.

The concrete case is `aggressor_side`. It is typed as a required `str` in `models/trade.py` and `models/feed_schema.py`, and the engine always publishes one of `BUY`, `SELL`, `AUCTION`. Yet four independent deserialisers default it to `""`, and `clearing/main.py` then writes `trade.aggressor_side or None` to undo that. Nobody decided `""` was legal; it accreted.

The spec therefore declares the strict contract and adds one lenient fallback:

```
      - name: aggressor_side
        type: enum
        values: [BUY, SELL, AUCTION]
        required: true
        parse_default: ""          # what from_dict substitutes; NOT a legal value
```

```
archived = {...}                                # no aggressor_side key
TradeExecuted.from_dict(archived).aggressor_side  # ""  — reads fine
TradeExecuted.from_dict(archived).validate()      # MessageValidationError
make_trade_executed(**archived)                   # MessageValidationError
```

Four consequences worth spelling out:

1. **Nothing that reads history breaks.** `from_dict` is a drop-in for the hand-written payload, byte-for-byte.
2. **Every published message is checked**, and the engine already always supplies a real value.
3. **"" never becomes a permanent part of the contract.** Had the spec declared it a legal enum value, it would have to become a C enum member — an invented `EDU_AGG_UNKNOWN` sentinel exporting the accident into a second language and freezing it in a wire format.
4. **The "" population becomes countable.** Run `validate()` over the clearing archive and read the failure count. Today nothing asserts, so nobody knows how many there are.

Which should I call?

- Reading data your system already published (replay, archive, audit, migration): `from_dict`, and `validate()` separately if you want to know. - Receiving a live message: `parse_*`. Fail at the boundary. - Publishing: `make_*`. Always. `MessageValidationError` subclasses `ValueError`, so existing call sites that already guard with `except ValueError` keep working unchanged.

11.14 The guarantee: `pm-msgen check`

Without a check, a generator is a scaffolder and the drift returns within a release. `pm-msgen check` re-renders every artefact from the spec and diffs it against what is committed. Any of these fails:

- a spec change without regenerating
- a hand-edit to a generated file
- a missing generated file

```
$ pm-msgen check
pm-msgen check: generated output is out of date with the spec.
Run `pm-msgen generate` and commit the result.

--- generated/trade.py (committed)
+++ generated/trade.py (from spec)
@@ -140,7 +140,7 @@
-    tick_decimals: int = 4
+    tick_decimals: int = 2
```

Determinism is not optional

The check only works if generation is byte-identical for an unchanged spec, run twice, on any machine. A generator that occasionally reorders its own output turns the check into a source of flaky CI failures — which is worse than not having it. The emitter therefore walks spec declaration order everywhere, never iterates a `set`, and puts no timestamp or absolute path in the banner. Five tests assert this, including one that regenerates under three different `PYTHONHASHSEED` values in separate processes, and one that generates twice to two directories and compares the files byte for byte.

11.14.1 Where it runs

Place	Command	Notes
`make check` / `make pre-commit`	`make msgen-check`	stamp-cached on `spec/*.yaml` and `src/**/*.py`, so it is free when nothing changed
CI, `code-check` job	`PYTHONPATH=src poetry run python -m edumatcher.msgen.cli check`	alongside black / flake8 / mypy

Why CI invokes the module, not the pm-msgen script

The `code-check` job installs with `install-root: 'false'` (`--no-root`), so the project's console scripts are not on `PATH` — only its dependencies are. `pyyaml` is a main dependency and *is* installed, so the module form works. "Simplifying" that step to `poetry run pm-msgen check` would fail with `command not found` instead of a drift report; `tests/test_msgen_ci_wiring.py` asserts against exactly that.
The wiring itself is tested. `tests/test_msgen_ci_wiring.py` parses the `Makefile` and `ci.yml` and fails if `msgen-check` drops out of `_check`, or if the CI step disappears. A guarantee that can be deleted by an unrelated refactor without anything noticing is not a guarantee.

11.15 Why the generated file looks the way it does

A few decisions that look odd until you know the reason:

It is already black-formatted, and black is never invoked. The emitter reproduces black's rules directly — double quotes, two blank lines around top-level definitions, a call exploded across lines only when the single-line form would exceed 88 columns. Running black at generation time would make the output depend on the installed black version, reintroducing exactly the flaky- check risk the previous section is about.

An enum is not always a Literal. `aggressor_side` is annotated `str`, not `Literal["BUY", "SELL", "AUCTION"]`, because its `parse_default` of `""` is outside those values — `from_dict` can legitimately produce it. Annotating it `Literal` anyway would make the type a lie that every call site has to silence with a `type: ignore`. Narrowing is `validate()`'s job. An enum *without* a non-conforming `parse_default` does get a proper `Literal`.

Regex patterns are module constants, interpolated by object. A validation message reads `{_TRADE_EXECUTED_ID_RE.pattern!r}` rather than embedding the pattern text, because a spec pattern may contain quotes or braces — either of which would break the f-string that carried it.

11.16 Migrating a family's topic literals

The codebase once carried **108 topic string literals across 25 files**. Each one is a place where a publisher-side rename goes unnoticed: the subscriber keeps compiling, keeps running, and simply stops receiving. Nothing errors.

`pm-msgen grep-literals` measures the remaining ones, per family:

```
$ poetry run pm-msgen grep-literals
trade: 26 literal(s) in 14 module(s)
    src/edumatcher/ai_trader/main.py:113: "trade.executed",
    src/edumatcher/ai_trader/main.py:273: if topic == "trade.executed":
    ...
26 literal(s) remaining across 14 module(s)
```

Migrating them is mechanical — the literal becomes the generated constant:

```
from edumatcher.models.generated.trade import TOPIC_TRADE_EXECUTED

sub = make_subscriber(ENGINE_PUB_ADDR, TOPIC_TRADE_EXECUTED, "book.")
...
if topic == TOPIC_TRADE_EXECUTED:
```

A family is migrated when its count reaches zero, and `tests/test_msgen_literals.py` is what keeps it there: add the family to that file's `MIGRATED` tuple and a reintroduced literal fails the suite rather than merging quietly. `trade` and `order` are at zero today.

Why the constant and not just a shared string

Both give you one definition. Only the generated constant is *checked against the spec* — rename the topic in `trade.yaml`, regenerate, and every subscriber picks it up; forget to regenerate and `pm-msgen check` fails the build. A hand-written `TRADE_TOPIC = "trade.executed"` in some constants module would drift from the spec exactly like the literals did.

11.17 Testing

File	Covers
`tests/test_msgen_spec.py`	the loader and its strictness — every rejection is a test
`tests/test_msgen_python.py`	determinism, drift detection, parameterised topics, every `validate` rule
`tests/test_msgen_trade_wire_compat.py`	wire compatibility between a producer and its generated builder
`tests/test_msgen_trade_perf.py`	hot-path budget (marker `perf`, deselected by default)
`tests/test_msgen_ci_wiring.py`	that the drift check is actually wired into the build
`tests/test_msgen_calf_roundtrip.py`	compiled C vs Python over the CALF wire (skips without `cc`)

File	Covers
tests/test_msgen_calf_adoption.py	what normalise_trade adoption changed, and what it did not
tests/test_msgen_balf_roundtrip.py	binary frames: byte-identity with an independent reference, compiled C round-trip, and the example-parser frame-size guard
tests/test_msgen_literals.py	a migrated family's topics appear nowhere as literals
tests/test_msgen_order_events.py	the five order events, the three presence regimes, and the one accepted wire change

Two tests in the wire-compat file are worth knowing about because they will fail if you change the wrong thing:

- test_no_trade_executed_literal_remains_in_adopted_modules scans the adopted modules for a "trade.executed" string literal. It is how a family's freedom from topic literals is enforced, and it catches a subscription literal left behind by mistake.
- test_key_order_differs_as_documented pins the two key orders against each other, so a producer's key order stays a recorded decision rather than becoming folklore.

11.18 Scope

What the generator owns: message shape, projection, construction, parsing and validation.

What it does not, and will not:

- **The wire formats.** It describes what already flows. CALF text and BALF binary are pedagogical artefacts that students read and parse by hand; replacing them with Protobuf would remove the teaching value.
- **Stateful normalisation.** normalise_trade now calls the generated projection for its {PX, QTY, SIDE} map, but the top-of-book cache it updates on every trade, and the delta suppression around it, stay hand-written — as do ralf_gateway's per-symbol execution counts. Drawing the line at "the generator owns the field map, the gateway owns the state" is what keeps this a message-shape tool rather than a second gateway.
- **How a client should render a value.** The generated struct gives calf_subscriber.c a typed double price, and that client still prints the raw wire string, deliberately: choosing a decimal count means knowing the instrument's tick scale. A typed binding removes guesswork about field names and types, not about market-data semantics.
- **Engine business logic**, and anything needing a validation language rich enough to express the risk rules — that would be a second implementation of the engine.

11.19 See also

- Message Reference — the hand-written narrative reference this generator will eventually supplement

CHAPTER 12

Container and Network Setup

Learning objectives

After reading this page you will understand:

- How the eight container images, the compose overlays and the three layers of Makefile fit together
- Where networking responsibility actually sits, and which of the several `127.0.0.1`/`0.0.0.0` knobs controls what
- What is baked into an image at build time versus decided at start time, and the precedence order between the ways of setting each
- How a release turns one git tag into a wheel and eight multi-architecture images
- What maintenance this design will demand as the system grows, and where it will drift first
- How to diagnose a container that "is running" but cannot be reached

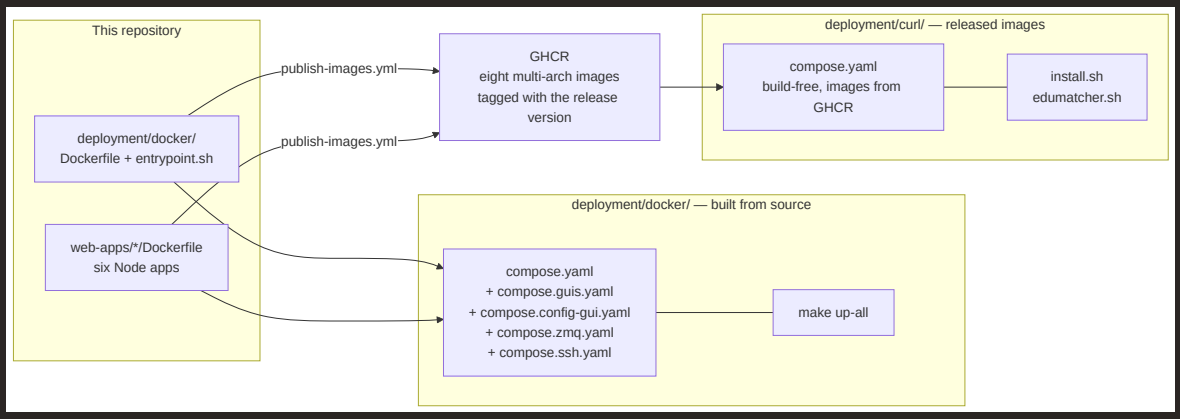
12.1 Summary

EduMatcher ships as **eight container images**. The backend and the five backend-connected GUIs run as **one Compose project** (`config-gui` is opt-in, and `edumatcher-ai-sim` runs in a project of its own):

Image	Built from	Serves
`edumatcher`	`deployment/docker/`	Every `pm-*` process — engine, gateways, REST API
`edumatcher-terminal-gui`	`web-apps/terminal-gui/`	TapeDeck (`pm-terminal`), port 8090
`edumatcher-log-gui`	`web-apps/log-gui/`	Log Operator Console (`pm-log-ui`), port 8091
`edumatcher-config-gui`	`web-apps/config-gui/`	Configuration GUI (`config-gui`), port 8092
`edumatcher-trader-gui`	`web-apps/trader-gui/`	Trading GUI (`pm-trading-ui`), port 8093
`edumatcher-book-gui`	`web-apps/book-gui/`	Order Book Viewer (`pm-book`), port 8094
`edumatcher-dash-gui`	`web-apps/dash-gui/`	Operator Dashboard (`pm-dash`), port 8095
`edumatcher-ai-sim`	`deployment/docker/`, stage `ai-sim`	The `edumatcher` image with the AI-trader simulation's defaults (`s300-sim`, profile `ai-sim`); run by `compose.ai-sim.yaml` with five of the GUIs, in a compose project of its own (The Simulation Container)

The backend is one container by design — the engine's ZeroMQ bus binds loopback by default, so every `pm-*` process must share a network namespace, with `pm-opctl-cli` as the process manager inside it. The GUIs are separate containers that reach it over the Compose network.

There are two deployments of that same set:



12.1.1 Compose file map

There are three families of compose files. Only the first is for development.

File	Role	Standalone?	Started by
`deployment/docker/compose.yaml`	Base: the one `edumatcher` backend container, ports, `./data` volume, bind hosts	Yes	`make up`, `make up-all`
`deployment/docker/compose.guis.yaml`	Overlay: the five backend-connected GUIs, built from `web-apps/`, in the **same project** so they reach `edumatcher:<port>`	No, needs the base	`make up-all`
`deployment/docker/compose.config-gui.yaml`	Overlay: `config-gui`, which needs no backend	No	`make up-all CONFIG_GUI=1`
`deployment/docker/compose.zmq.yaml`	Overlay: publish the raw ZeroMQ bus (5555-5559, 5601/5602, 5611)	No	`ZMQ=1`
`deployment/docker/compose.ssh.yaml`	Overlay: sshd on `SSH_PORT` (default 2222)	No	`SSH=1`
`deployment/docker/compose.ai-sim.yaml`	The AI-trader simulation and five GUIs, images from GHCR (the backend can be built locally with `LOCAL=1`)	Yes, ignores `.env`	`make up-sim`
`deployment/curl/compose.yaml`, `compose.zmq.yaml`, `compose.ai-sim.yaml`	Released twins of the files above, build-free, images from GHCR	Same as the twin; `compose.yaml` already includes the GUIs	`edumatcher.sh`
`deployment/curl/compose.guis.yaml`	**Only** the six GUIs, for an exchange running on the host (pipx install). Despite the name this is *not* a twin of the docker overlay	Yes	`edumatcher.sh` with `EM_STACK=guis`

File	Role	Standalone?	Started by
`web-apps/<app>/docker-compose.yml` (+ `docker-compose.linux.yml`)	One GUI alone, talking to a backend **outside** compose via `host.docker.internal` (the Linux overlay adds the `host-gateway` mapping)	Yes	`make up` in the app, or `make up` in `web-apps/`

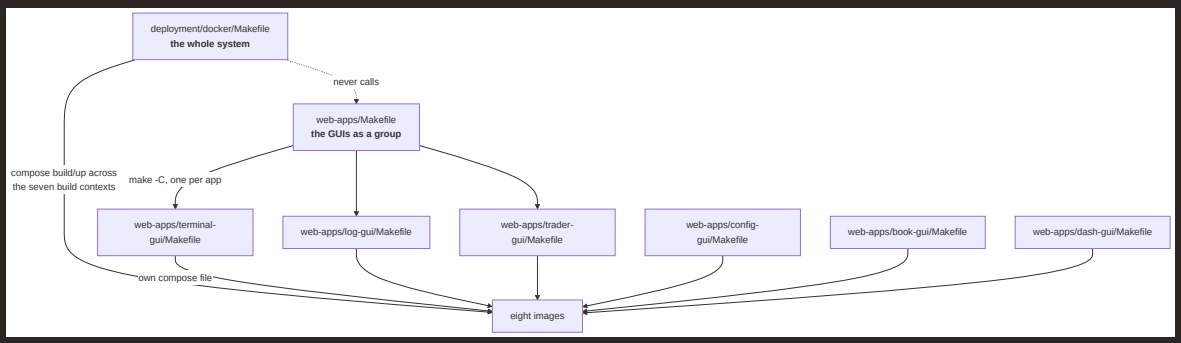
The decisive difference is networking. In the `deployment/` families every container shares one compose project, so the GUIs find the backend by service name. The per-app files have no such project, so they reach the backend through the host, which is why they need `*_HOST` overrides on Podman or in a VM.

Overlays are additive `-f` arguments. The Makefile builds the list from its flags, so you rarely type them: `make up-all ZMQ=1 CONFIG_GUI=1` runs `compose -f compose.yaml -f compose.zmq.yaml -f compose.guis.yaml -f compose.config-gui.yaml up -d`. `make -n up-all ZMQ=1` prints the exact command.

`deployment/docker/` is the one you use while changing the code: it builds from your working tree. `deployment/curl/` is its released twin: identical wiring, no `build:` sections, images pulled by tag. Keeping those two in step is the main standing maintenance cost of this design — see Part 6.

12.2 Part 1 — How the Makefiles are structured

There are three layers, and they answer three different questions.



Layer	File	Owns	Use it when
System	`deployment/docker/Makefile`	The backend image, the web-app images (`build-guis`), and Compose across all six services. Data directory, profiles, configuration deployment, publishing	You want a working exchange with GUIs
Group	`web-apps/Makefile`	Fanning out to three GUI Makefiles with a shared `VM_BACKEND_IP`	You run the GUIs against a backend that is *not* in the same Compose project — a Multipass VM, or a host install
App	`web-apps/<gui>/Makefile`	One application: npm workspace, its own image, its own compose file	You are developing that one GUI

The system layer does **not** call the group layer. It reaches the same Dockerfiles directly through `compose.guis.yaml`'s build contexts (`../../web-apps/<gui>`). That is deliberate — one Compose project is what puts every container on one network — but it does mean the GUI service definitions exist in two places. See Part 6.

12.2.1 What the app Makefiles give you

Each GUI Makefile has the same shape (`make help` in any of them):

Group	Targets
npm workspace	`install`, `build`, `build-debug`, `typecheck`, `test`, `lint`, `format`
Development servers	`dev`, `dev-web`, `dev-bridge`
Container	`cbuild`, `up`, `down`, `restart`, `logs`, `ps`
Distribution	`cdist` — an image exported as an `xz`-compressed tarball for offline delivery
Registry	`ghcr-login`, `cpush`, `ghcr-logout`

The development servers are the interesting ones: `make dev` runs each app's Vite server and its bridge with hot reload, against the exchange running in containers. That inner loop — and the fact that `make up-all` will *not* pick up a web-app change on its own — is The Development Loop.

Two different GHCR paths exist, and they tag differently

A GUI's own `make cpush` tags with **that app's `package.json` version** (terminal-gui is `0.1.0`), while `deployment/docker`'s `make ghcr-push` and the release workflow tag with the **EduMatcher release version** (`0.20.5`). Only the latter produces a coherent set that the installer can pin with one number. Use a per-app `cpush` for ad-hoc work on one GUI; never as part of a release.

12.2.2 Variable names that mean different things

Two collisions are worth committing to memory before you invent a third:

Name	In `deployment/docker/Makefile`	Elsewhere
`VERSION`	Which **PyPI release to install into the image** (`make build VERSION=0.20.4`)	In a GUI Makefile, that app's `package.json` version
`TAG`	Which **registry tag to push** (`make ghcr-push TAG=0.20.6`)	—
`CONFIG`	A `make`-level flag: an example name *or* a file path	`EM_CONFIG` / `EM_CONFIG_FILE` are the container-level variables it sets

`ghcr-push` uses `TAG` rather than `VERSION` precisely because `VERSION` already had a contradictory meaning one target away.

12.3 Part 2 — Networking: who is responsible for what

12.3.1 Bind address versus connect address

A ZeroMQ or TCP endpoint appears twice in this codebase and the two are not interchangeable:

- A **bind** address is what a server listens on. `127.0.0.1` accepts only connections originating inside the same network namespace. `0.0.0.0` accepts connections arriving on any interface the namespace has.
- A **connect** address is what a client dials. It must name a host that is reachable *from the client's* namespace.

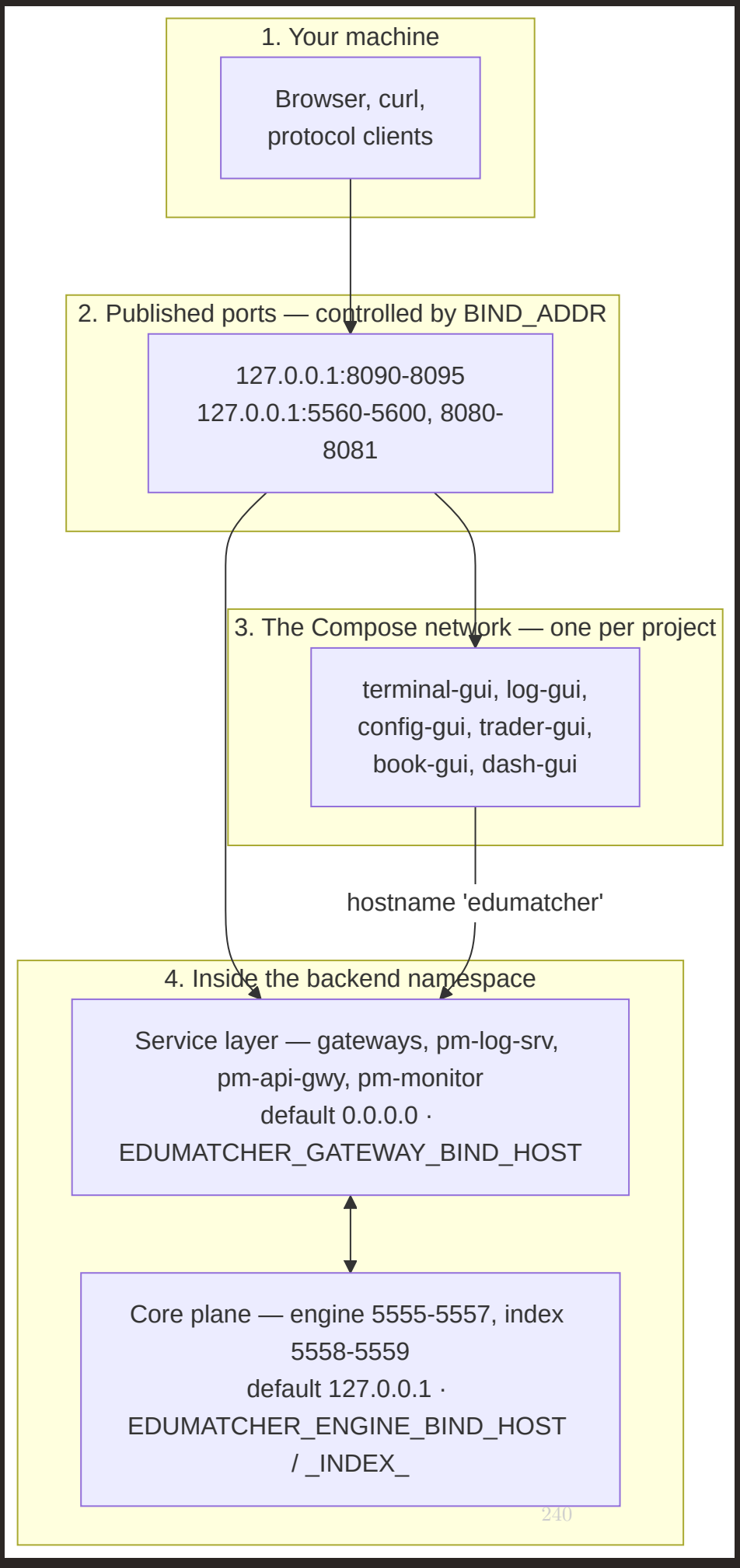
In `edumatcher/config.py` these are separate constants:

```
# Bind side — where a server listens
EDUMATCHER_ENGINE_BIND_HOST = os.getenv("EDUMATCHER_ENGINE_BIND_HOST", "127.0.0.1")
ENGINE_PULL_BIND_ADDR = f"tcp://{EDUMATCHER_ENGINE_BIND_HOST}:5555"

# Connect side — where a client dials
EDUMATCHER_ENGINE_HOST = os.getenv("EDUMATCHER_ENGINE_HOST", "127.0.0.1")
ENGINE_PULL_ADDR = f"tcp://{EDUMATCHER_ENGINE_HOST}:5555"
```

Setting the bind host does not move any client, and setting the connect host does not open any listener. Cross-host work needs both.

12.3.2 The four layers an address passes through



Layer	Controlled by	Set where	Default	`0.0.0.0` here means
Published ports	`BIND_ADDR`	`.env`	`127.0.0.1`	**Anyone on your LAN can reach the exchange**
Compose network	Compose project membership	which `-f` overlays you pass	—	—
Service layer	`EDUMATCHER_GATEWAY_BIND_HOST`; then `bind_address:` / `host:`	`compose.yaml`; `engine_config.yaml`	`0.0.0.0`	Any container on the Compose network, and anything reaching a published port
Core plane	`EDUMATCHER_ENGINE_BIND_HOST`, `EDUMATCHER_INDEX_BIND_HOST`	`compose.zmq.yaml`, via `ZMQ=1`	`127.0.0.1`	Same, plus publishable to the host

Only the first is a security boundary. The rest are inside a namespace that already isolates the container; widening them exposes nothing that a published port does not already expose.

12.3.3 Why the GUIs need no host address at all

All the services belong to one Compose project, so Compose puts them on one network and registers the backend under the service name, the container name *and* the hostname — all three resolve to the same address:

```
$ podman exec edumatcher-trader-gui getent hosts edumatcher
10.89.2.2       edumatcher.dns.podman
```

That is why `compose.guis.yaml` points every GUI at `edumatcher:<port>` and why `host.docker.internal`, the `host-gateway` pseudo-address and `VM_BACKEND_IP` have all disappeared from that path. Those existed to route GUI traffic out to the host and back in, which behaved differently on Docker Desktop, Linux Docker and Podman. `web-apps/Makefile` still carries `VM_BACKEND_IP` because it serves the other case: GUIs in one project talking to a backend that is somewhere else entirely.

12.3.4 The two planes and their defaults

Every listening socket the backend opens belongs to one of two planes, and the asymmetry between them is deliberate rather than historical.

	Core plane	Service layer
Processes	`pm-engine`, `pm-index`	`pm-alf-gwy`, `pm-md-gwy`, `pm-balf-gwy`, `pm-ralf-gwy`, `pm-dc-gwy`, `pm-log-srv`, `pm-api-gwy`, `pm-monitor`
Ports	5555–5559	5560, 5565, 5570, 5580, 5590, 5600, 5601/5602, 5610/5611, 8080/8081
Who dials in	Other `pm-*` processes only	GUI containers, student clients, `curl`, Swagger
Default bind host	`127.0.0.1`	`0.0.0.0`
Constant	`EDUMATCHER_ENGINE_BIND_HOST`, `EDUMATCHER_INDEX_BIND_HOST`	`EDUMATCHER_GATEWAY_BIND_HOST`
Opened by	`ZMQ=1` / `EM_ZMQ=1`	already open

`pm-monitor` is the one service-layer process whose own default is `127.0.0.1` (`monitor.bind_address`): every process that sends it a beacon runs in the same namespace. The compose file's `EDUMATCHER_GATEWAY_BIND_HOST=0.0.0.0` still opens it to the Compose network, which is how the operator dashboard reaches its snapshot port 5611. Its ports are `EXPOSE`d by the image but published by no compose file.

The core plane is an internal bus. Every peer shares the backend's namespace, so loopback is both sufficient and the safer default; `ZMQ=1` widens it when you want `pm-calf-spy` or `pm-index-cli` attaching from the host. The service layer is the product's

public surface — a listener nobody outside the namespace can reach is a listener that does not do its job — so it defaults open, and the namespace plus `BIND_ADDR` provide the isolation.

12.3.5 How a service listener resolves its bind host

All seven service processes share one resolver in `edumatcher/config.py`:

```
EDUMATCHER_GATEWAY_BIND_HOST = os.getenv("EDUMATCHER_GATEWAY_BIND_HOST") or None
DEFAULT_GATEWAY_BIND_HOST = "0.0.0.0"

def resolve_gateway_bind_host(configured: str | None = None) -> str:
    if EDUMATCHER_GATEWAY_BIND_HOST:
        return EDUMATCHER_GATEWAY_BIND_HOST
    return configured if configured else DEFAULT_GATEWAY_BIND_HOST
```

It is wired in at two points in each service's `config.py`, and both are needed:

```
# 1. The dataclass default — used when no configuration file supplies the key
bind_address: str = field(default_factory=resolve_gateway_bind_host)

# 2. The loader — used when one does
bind_address = resolve_gateway_bind_host(section.get("bind_address"))
```

`field(default_factory=...)` rather than a plain default because the environment must be read when the object is constructed, not when the module is imported into the test suite. `pm-api-gwy` spells the key `host:` instead of `bind_address:`; everything else about it is identical.

Precedence, first match wins:

```
--bind / --host flag  →  EDUMATCHER_GATEWAY_BIND_HOST  →  bind_address:/host: in YAML  →  0.0.0.0
```

The environment variable sits **above** the configuration file, which is the opposite of the usual ordering in this project and is the point: it is a deployment-wide switch that closes or redirects every service listener at once without editing a configuration that may belong to someone else. Both `compose.yaml` files set it to `0.0.0.0` explicitly — redundant against the code default, but it means a user-supplied configuration that pins loopback still comes up reachable from the GUI containers. It is written as `${EDUMATCHER_GATEWAY_BIND_HOST:-0.0.0.0}`, so an operator can still pin one interface from `.env`.

`src/edumatcher/config_show/extract.py` calls the same resolver, so `pm-config-show` and `make config-show` report the host each process will actually bind rather than the literal in the YAML.

12.3.6 Why the entrypoint no longer rewrites anything

Until v0.27.0 the bundled examples set `bind_address: 127.0.0.1` and `entrypoint.sh:open_gateway_binds()` rewrote loopback binds to `0.0.0.0` in the deployed YAML, then re-ran `pm-config-deploy`. It worked, but it had three costs: the deployed configuration was not the one the user wrote, the rewrite was invisible unless you read the container log, and `pm-config-show` described the file rather than the behaviour.

The defaults now carry that responsibility. Every bundled example says `0.0.0.0`, `open_gateway_binds()` and `LOOPBACK_BIND_RE` are gone from `entrypoint.sh`, and the deployed configuration is byte-for-byte the one supplied.

This is a behaviour change for bare-metal installs

A `pipx` or Poetry install has no namespace around it, so service listeners that used to default to loopback are now on every interface. `EDUMATCHER_GATEWAY_BIND_HOST=127.0.0.1` restores the old behaviour for a whole install in one variable. See `CHANGELOG.md`.

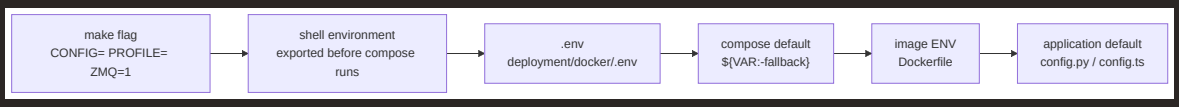
12.4 Part 3 — Build time versus run time

The single most common way to waste an hour here is to change something at run time that was decided at build time, or the reverse.

Decided at **build** time (baked into the image)	Decided at **run** time (changeable with a restart)
Which EduMatcher package is installed — local wheel, PyPI, or a pinned version	Which engine configuration is deployed (`EM_CONFIG`, `EM_CONFIG_FILE`)
Whether `openssh-server` exists (`WITH_SSH`)	Whether `sshd` is started (`EM_SSH`, via `SSH=1`)
The Python base image (`PYTHON_VERSION`)	Which process profile runs (`EM_PROFILE`)
`EDUMATCHER_DATA_DIR=/data`, `PATH`, the entrypoint	Timezone (`TZ`)
The compiled SPA bundle for each GUI	Which host interface ports publish on (`BIND_ADDR`)
npm registry, proxy and CA settings used during the build	Every GUI's backend addresses and API key
Each GUI's default `WEB_PORT`	The port it actually listens on (`PORT`), and its host mapping

The one that catches people is the SPA bundle: a change to any GUI's frontend needs an image rebuild. `make up-all` builds an image that is missing, but does not rebuild one that already exists — use `make build` or `$(COMPOSE) ... build` when you have changed frontend source.

12.4.1 Precedence when the same thing is settable in several places



Leftmost wins. A `make` flag becomes an exported variable for that one invocation only; `.env` is Compose's own file and is where machine-specific choices belong; the `${VAR:-fallback}` defaults inside the compose files are the last stop before the image's own `ENV`.

This is why `make up-all CONFIG=s10-nominal` does not edit `.env`, and why re-running plain `make up-all` afterwards goes back to whatever `.env` says. Compose notices the changed environment and recreates the container, so the switch takes effect without an explicit `down`.

12.4.2 Build arguments

Backend (`deployment/docker/Dockerfile`): `PYTHON_VERSION`, `EDUMATCHER_VERSION`, `PIP_INDEX_URL`, `WITH_SSH`.

Each GUI (`web-apps/*/Dockerfile`): `WEB_PORT`, `NPM_REGISTRY`, `NPM_STRICT_SSL`, `HTTP_PROXY`, `HTTPS_PROXY`, `NO_PROXY`; config-gui also takes `USE_PROXY_CA` and `CA_CERT_FILE`, and log-gui `EDUMATCHER_VERSION` for the Python package it bundles.

The backend's package source is selected by what is present rather than by a flag:

```
COPY .wheel/ /tmp/wheel/
RUN if ls /tmp/wheel/*.whl; then pip install /tmp/wheel/*.whl;   # local build
    elif [ -n "${EDUMATCHER_VERSION}" ]; then pip install "edumatcher==${EDUMATCHER_VERSION}";
    else pip install edumatcher; fi                              # newest on PyPI
```

`make build` puts a freshly built wheel in `.wheel/`; `PYPI=1` and `VERSION=x.y.z` clear it first. log-gui installs the same package the same way from `web-apps/log-gui/.wheel/` (its Diagnostics view runs `pm-log-cli`), and `make build-guis` and `make ghcr-push` fill that drop box by the same rule.

Always confirm which package went into the image

A successful build prints `Installing local wheel: /tmp/wheel/edumatcher-<version>.whl`. If you see pip *downloading* `edumatcher` instead, the image contains a published release and your source change is not in it — while everything else about the build looks fine. This exact failure mode has cost real debugging time on this project, from a path in the `RUN` step that did not match the `COPY` destination.

12.5 Part 4 — Starting and configuring the whole system

```
cd deployment/docker
make build          # backend image from a wheel built out of this checkout
make up-all         # exchange + the five backend-connected GUIs
make down-all       # stop and remove everything
```

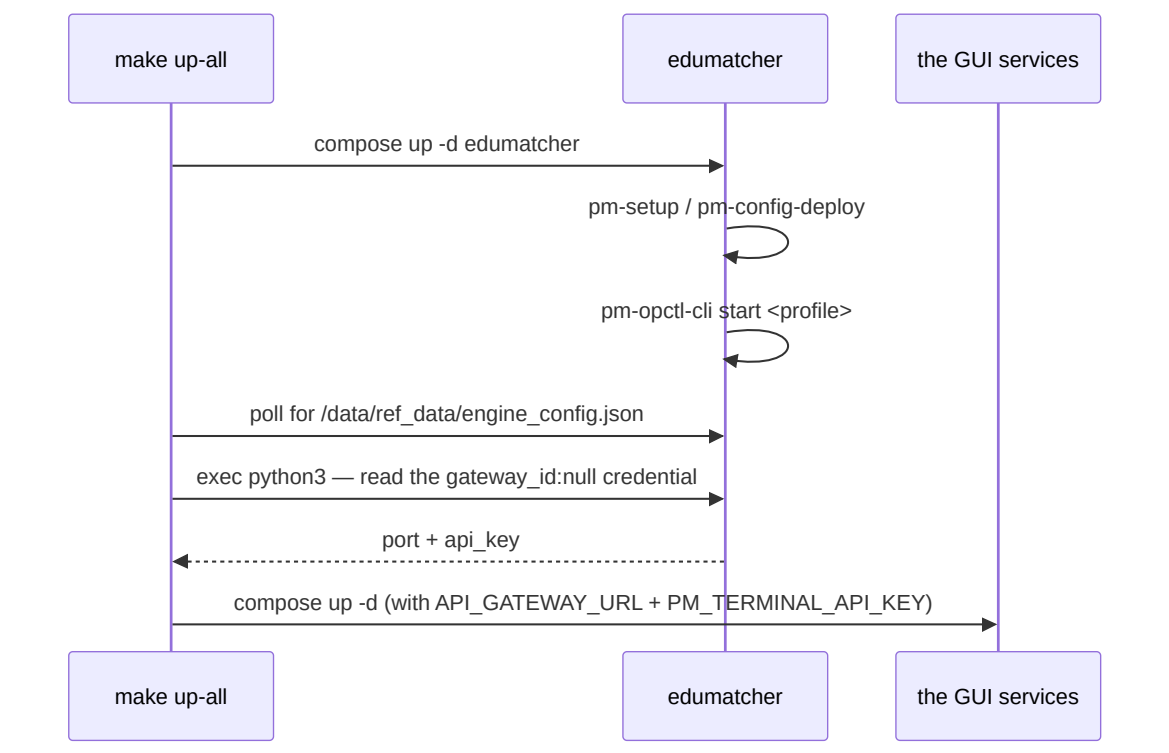
Flag	Applies to	Effect
`CONFIG=<name>`	`up`, `up-all`	Deploy a bundled example
`CONFIG=<file>`	`up-all`	Copy the file to `config/`, mount it read-only at `/config`, deploy it
`PROFILE=<name>`	`up`, `up-all`	`default`, `mini`, `micro`, `mm-demo`, `ai-swarm` or `ai-sim`
`ZMQ=1`	`up`, `up-all`	Add `compose.zmq.yaml`: publish the raw bus and set the engine/index bind hosts to `0.0.0.0`
`SSH=1`	`up`, `up-all`	Add `compose.ssh.yaml` and prepare `authorized_keys`
`CONFIG_GUI=1`	`up-all`	Add `compose.config-gui.yaml`
`PYPI=1`, `VERSION=`	`build`	Install from PyPI instead of the checkout
`CONTAINER_ENGINE=docker`	all	Force Docker when Podman is also installed

`make mounts` reports, for every container, which host directory is behind each path inside it and which image it came from — the fastest way to tell whether the stack answering your ports is the one you started.

The overlays are additive `-f` files, which is why they compose freely: `make up-all CONFIG=s10-complex PROFILE=mini ZMQ=1 CONFIG_GUI=1`.

12.5.1 Why `up-all` runs in two phases

TapeDeck reads history through `pm-api-gwy` using the credential whose `gateway_id` is `null`. That key is **generated per engine configuration** — different in each example — and is issued on the `dashboards` instance (8081), not `desk` (8080). It does not exist until the exchange has deployed its configuration, so it cannot be a compose default.

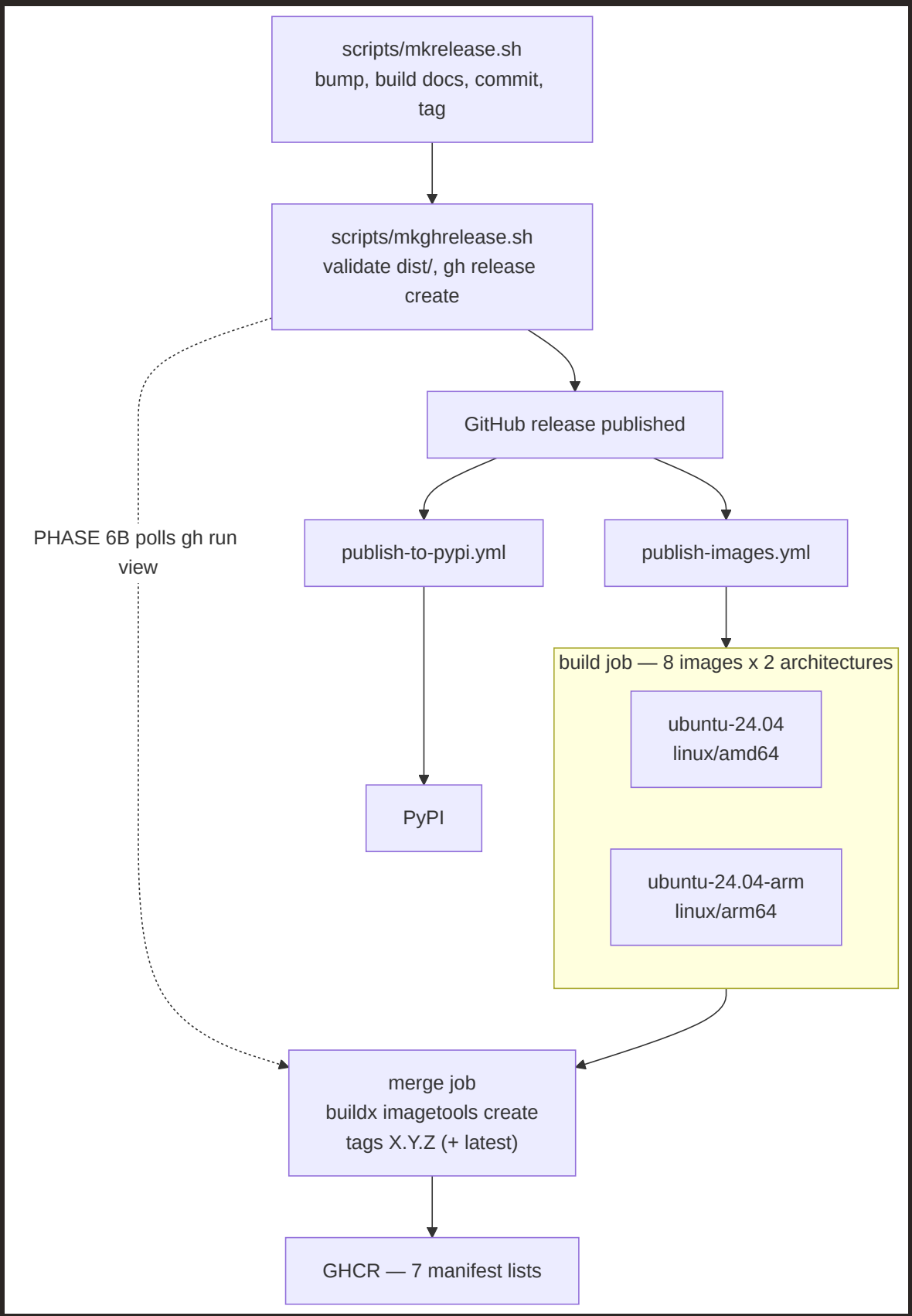


`deployment/curl/edumatcher.sh start` performs the identical sequence with the same extraction snippet — that logic lives in `edumatcher.sh` rather than in `install.sh` so there is one copy on the released path.

A configuration with no read-only credential is not an error: the start continues and warns. The live CALF feed needs no key; only history does.

12.6 Part 5 — The release process

One git tag produces a wheel and eight multi-architecture images, all carrying the same version. That coupling is what lets `install.sh --version 0.76.5` pin an entire system with one number.



Points a developer needs to know:

Each image is built twice, natively. A single-runner build with `platforms: linux/amd64,linux/arm64` would emulate the foreign architecture under QEMU — six Node images each running `npm ci` and a Vite build, which turns a release into hours. Instead each `(image, arch)` pair builds on its own native runner and pushes **by digest only**; a merge job then joins the two digests into a manifest list and applies the tags. arm64 hosted runners are generally available and free for public repositories.

The merge job selects digests by exact image prefix. Each build job uploads its digest as an artifact named `digests-<image>-<arch>`, and the merge job collects them with a glob. `edumatcher` is a prefix of all six `edumatcher-*-gui` names, so that glob over-matches: the backend's merge job receives every image's digests. It therefore filters the downloaded files — each is named `<image>@<hex>` — by exact prefix, and asserts it ended up with exactly two. Without the filter the backend asks the registry for a GUI's digest under `ghcr.io/…/edumatcher` and gets `not found`; without the count check, a missing architecture would publish a single-architecture image under the release tag, which nobody notices because the maintainer's own machine still works.

The backend's wheel is built inside the workflow, not taken from PyPI. `publish-to-pypi.yml` fires on the same `release: published` event, so whether the new version has reached PyPI yet is a race — and losing it means shipping an image one release behind. The workflow runs `poetry build` and drops the wheel into `deployment/docker/.wheel/` before the image build.

`latest` only moves for an exact `vMAJOR.MINOR.PATCH` tag, so a pre-release never becomes what a new user gets by default. That is the same test `publish-to-pypi.yml` uses to choose between PyPI and TestPyPI.

`mkghrelease.sh` waits but does not push. Its PHASE 6B polls `gh run view` for the image workflow — bounded by `IMAGE_WAIT_MINUTES` (default 30), skippable with `--skip-images` — and reports the eight image references. If the workflow fails the GitHub release still exists; only the images are missing, and `gh workflow run publish-images.yml -f tag=vX.Y.Z` re-runs just that part.

12.6.1 The manual escape hatch

`make ghcr-push` in `deployment/docker/` builds all six images from your checkout and pushes them. It exists for when the workflow cannot run at all.

```
export GITHUB_USER=<you> GHCR_TOKEN=<token with write:packages>
make ghcr-push                              # all six, tagged :dev
make ghcr-push TAG=0.20.6 FORCE=1 LATEST=1  # as a release tag
```

It can only build for the architecture you are on. Pushing single-arch over a release tag **replaces the manifest list**, and users on the other architecture then get "no matching manifest" — a failure you will not observe, because your own machine still works. Hence `FORCE=1` for a release-looking tag, and `latest` never moving unless asked.

12.6.2 Two release failures that are silent

Failure	Who sees it	What catches it
Image built from PyPI instead of the checkout	Nobody — it looks like a normal build	The `Installing local wheel:` line in the build output
A tag published with only one architecture	Only users on the other architecture	The merge job's two-digest assertion
GHCR packages left private	Everyone except the maintainer, who is already authenticated	Installing from a machine with no credentials
A package that predates the workflow	Only that one image, and only in CI	`permission_denied: read_package` on push

A newly created GHCR package is private. Until each of the eight is made public in its package settings, `podman pull` fails for every user but you.

12.6.3 A package the workflow did not create

A package that a workflow pushes for the first time is linked to the repository automatically, and `GITHUB_TOKEN` can write to it from then on. A package that already existed — because someone pushed it by hand with a personal access token, for instance with a GUI's own `make cpush` — belongs to the *user* account and has no repository in its access list. `GITHUB_TOKEN` is scoped to the repository, so the push fails:

```
#21 [auth] johan162/edumatcher-config-gui:pull,push token for ghcr.io
#21 DONE 0.0s
#20 pushing layers 0.5s done
#20 ERROR: failed to push ghcr.io/johan162/edumatcher-config-gui: denied: permission_denied: read_package
```

The authentication *succeeds* and the layers upload; only the manifest write is refused. Fix it once, on the package's page under **Package settings → Manage Actions access → Add repository**, granting the repository the **Write** role. Deleting the package and letting the workflow recreate it works too, and loses whatever was published under it.

This is another failure that only appears for images somebody published by hand before the workflow existed — which is one more reason not to use a per-app `cpush` as part of a release.

12.7 Part 6 — Maintenance burden

This design trades some duplication for a property that is hard to get any other way: one Compose project, therefore one network, therefore no host addresses anywhere. Knowing exactly what that costs is the point of this section.

12.7.1 Adding a sixth application

Every place that has to change, in order:

1. `web-apps/<new-gui>/` — Dockerfile, compose file, Makefile (copy the closest existing app; they are deliberately near-identical)
2. `deployment/docker/compose.guis.yaml` — service definition pointing at `edumatcher:<port>`, **never** at a host address
3. `deployment/curl/compose.yaml` — the same service, image from GHCR, no `build:` section
4. `deployment/docker/Makefile` — add the image to `GUI_IMAGES`
5. `.github/workflows/publish-images.yml` — add it to **both** matrices, the `build` job's `image:` list and the `merge` job's. If the new name is a prefix of another image's name, or another's is a prefix of it, the digest filter in the merge job already handles it — but check that the count assertion still expects one digest per architecture
6. `deployment/docker/.env.example` and `deployment/curl/.env.example` — its host port
7. `deployment/curl/edumatcher.sh` — the `urls` output
8. Docs — the port tables in `../../operator-guide/part-1-install-and-deploy/010-installation.md`, `deployment/docker/README` and `deployment/curl/README.md`

Steps 2 and 3 are the pair that will drift.

12.7.2 Known duplication, and how to detect drift

Duplicated	Between	Detection
GUI service definitions	`deployment/docker/compose.guis.yaml` and `deployment/curl/compose.yaml`	The script below
The read-only-credential lookup	`deployment/docker/Makefile`, `deployment/curl/edumatcher.sh`, `web-apps/terminal-gui/Makefile`	Three copies in two languages; a change to the credential schema touches all three
The image list	`Makefile:GUI_IMAGES` and both workflow matrices	A missing entry means an image silently is not published
Port numbers	Both `.env.example` files, both compose files, three READMEs, the Operator's Guide installation chapter	Grep

A drift check for the compose pair, worth running before a release:

```
python3 - <<'PY'
import yaml
dev = {**yaml.safe_load(open('deployment/docker/compose.yaml'))['services'],
       **yaml.safe_load(open('deployment/docker/compose.guis.yaml'))['services'],
       **yaml.safe_load(open('deployment/docker/compose.config-gui.yaml'))['services']}
rel = yaml.safe_load(open('deployment/curl/compose.yaml'))['services']
assert sorted(dev) == sorted(rel), (sorted(dev), sorted(rel))
keys = ["CALF_HOST","CALF_PORT","API_GATEWAY_URL","LOG_SRV_HOST","LOG_SRV_PORT",
        "LOG_SRV_PUB_PORT","LOG_SRV_PULL_PORT","API_PROXY_TARGET","LOG_DB_PATH"]
for svc in sorted(rel):
    d, r = dev[svc].get('environment', {}), rel[svc].get('environment', {})
    for k in keys:
        assert d.get(k) == r.get(k), f"{svc}.{k}: dev={d.get(k)!r} release={r.get(k)!r}"
print("compose files agree")
PY
```

12.7.3 Standing costs

Version coupling. Every image carries the EduMatcher release version, so every release republishes all eight even when only the backend changed. That is the price of "one tag names a coherent set" and is almost certainly the right trade — but it means a GUI-only fix still needs a full release.

The GUI `package.json` versions are now decorative for release purposes. They still drive `make cdist` and per-app `cpush`. If that divergence starts causing confusion, the fix is to stop versioning the apps independently rather than to reintroduce per-app tags into the release.

Build minutes grow linearly with images. Sixteen build jobs today; one more image makes it eighteen. The GitHub Actions cache is scoped per `(image, arch)`, so an unchanged app is cheap, but the matrix is not free.

The credential-lookup snippet is the fragile one. It reads a specific shape out of the compiled configuration — `api_gateways.<name>.credentials[].gateway_id`. A schema change there breaks three call sites that no test covers.

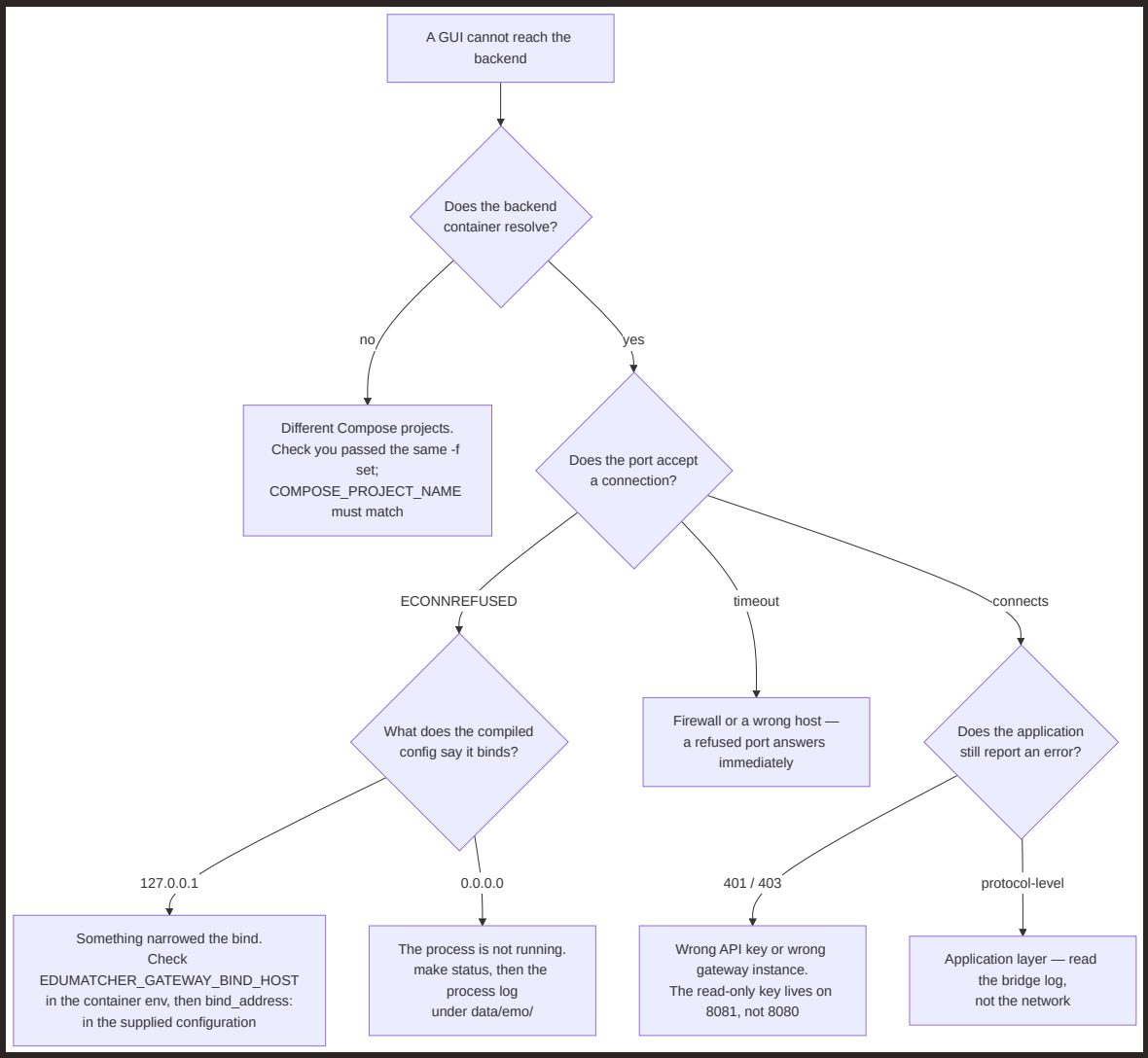
12.8 Part 7 — Troubleshooting

12.8.1 Two signals that look like proof and are not

"I can reach the port from the host." Podman's rootless port forwarding delivers a published connection to *loopback inside the container's namespace.* A gateway bound to `127.0.0.1` therefore answers host traffic perfectly while refusing every sibling container. Host reachability tells you nothing about container-to-container reachability. The defaults no longer produce this, but `EDUMATCHER_GATEWAY_BIND_HOST`, a `--bind`/`--host` flag or a `bind_address:` in a supplied configuration still can.

"`make status` is green." `pm-opctl-cli list` reports `tcp connect to 127.0.0.1:5570 ok`, but that probe also runs inside the container over the same loopback. A healthy process table is not evidence that anything outside that container can connect.

12.8.2 Tracking down a network block



The ladder of commands behind that tree:

```
# 1. Name resolution across the Compose network
podman exec edumatcher-trader-gui getent hosts edumatcher

# 2. An actual connection attempt, from where it matters — with a
#    known-good port as the control
podman exec edumatcher-terminal-gui node -e '
const net = require("net");
for (const p of [5600, 5570]) {
  const s = net.connect(p, "edumatcher", () => { console.log(p, "OPEN"); s.end(); });
  s.on("error", e => console.log(p, "FAIL", e.code));
}'

# 3. What the processes actually read — the compiled artifact, not the YAML
podman exec edumatcher python3 -c \
  "import json; print(json.load(open('/data/ref_data/engine_config.json'))['market_data_gateway'])"

# 4. What bind host the processes were actually told to use
podman exec edumatcher env | grep -E 'EDUMATCHER_(GATEWAY|ENGINE|INDEX)_BIND_HOST'
podman exec edumatcher pm-config-show --section ports

# 5. What the bridges think of their own uplinks
curl -s localhost:8090/api/bridge/status   # calf, logging
curl -s localhost:8091/api/bridge/status   # lalfPs, logDb
curl -s localhost:8094/api/bridge/status   # upstream, logging
curl -s localhost:8095/api/bridge/status   # pm-monitor, metricsDb, links

# 6. Which ports are actually published
podman port edumatcher

# 7. Which directory on disk is behind each container path, and where each
#    container came from.  make mounts  in deployment/docker, or
#    ./edumatcher.sh mounts  in a released install. The raw forms:
podman inspect edumatcher-log-gui \
  --format '{{range .Mounts}}{{.Source}} -> {{.Destination}}{{"\n"}}{{end}}'
podman ps --filter name=edumatcher --format '{{.Names}}\t{{.Image}}\t{{.CreatedAt}}'
```

Step 3 is the one that settles arguments. The deployed YAML in `ref_data/` is the *source* copy; the compiled JSON beside it is what every process reads, with defaults resolved. Reading the YAML has sent people down the wrong path on this project more than once.

12.8.3 Other things that will bite

Symptom	Cause
`terminal-gui` shows a live book but empty history	`PM_TERMINAL_API_KEY` unset, or `API_GATEWAY_URL` pointing at 8080 where the read-only key is not valid
`log-gui` shows live entries but no history	The `./data:/backend-data:ro` mount is missing or empty; the viewer needs the *file* `log.db`, not just the socket
A source change has no effect	The image was built from PyPI — see the warning in Part 3
`no container with name or ID ... found` during `down`	podman-compose removes only what the named files declare; Docker Compose removes by project label and does not care. `down-all` adds the config-gui overlay only when that container exists

Symptom	Cause
A GUI shows data that predates this install	Another install's containers. `make mounts` / `./edumatcher.sh mounts` names the directory behind each container path. Both deployments use the same fixed container names and host ports, so compose leaves an existing one alone and serves you its exchange. `podman ps` — the image tells you which: `ghcr.io/...` is the released stack, `localhost/...` the source-built one. Both `start` paths now refuse this rather than attaching
No health status in `podman ps`	Podman ignores the image `HEALTHCHECK` on OCI-format images. Nothing here depends on it — the Makefile sequences the phases itself
`apt-get` fails with "Release file is not valid yet"	The podman-machine VM's clock has drifted behind after a host suspend. The Dockerfile passes `Acquire::Check-Date=false`, but fix the clock: it also skews the trading calendar and every timestamp in `log.db`

12.9 Part 8 — What a genuine multi-host split would still need

Nothing in this section is implemented. It exists so the next person to take the deployment in that direction knows what is already in place.

12.9.1 What exists today

- The engine can bind a real interface, not only loopback (`EDUMATCHER_ENGINE_BIND_HOST`), and `pm-index` follows the same pattern.
- Every consumer resolves the engine through `EDUMATCHER_ENGINE_HOST`, so pointing a remote gateway or viewer at another host needs no source change.

12.9.2 What is missing

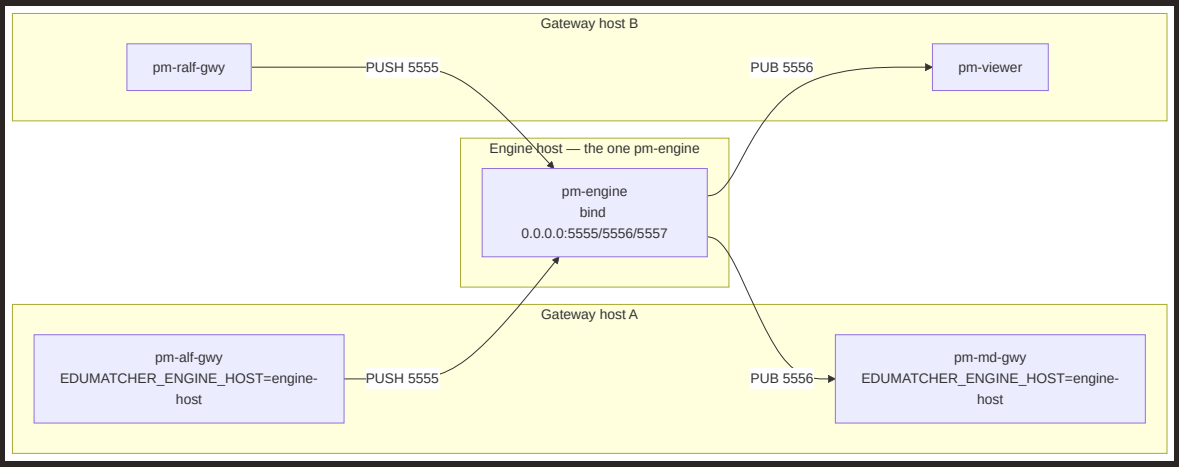
Load balancing is not possible, because there is exactly one engine. `pm-engine` owns the order book, the trade sequence and the single source of truth. Multi-host here can only mean distributing *gateways and viewers* across hosts, all still talking to one engine. Settle that scope question before writing code.

No transport security or authentication. ZeroMQ PUSH/PULL and PUB/SUB accept whatever connects. The loopback bind was an accidental safety net; making the bind host configurable removed it. Participants authenticate to the ALF/BALF *gateways*, and the engine trusts whatever a gateway forwards — an assumption that stops being free once port 5555 is reachable across a real network. TLS, ZeroMQ CURVE, or network ACLs are the options; none are implemented.

No service discovery. Every connect address is one static environment variable naming one host.

No cross-host orchestration. `compose.yaml` describes containers on one host. Several hosts means running the engine independently on each and wiring the variables by hand.

No cross-host health signalling. `pm-opctl-cli health`'s built-in recorder entries — `stats`, `audit`, `clearing` and `index-srv` — run each process's own `*-cli health -q`, which pings a health socket on loopback (5561–5564); the `monitor` entry runs `pm-monitor-cli health -q`, which waits for one of `pm-monitor`'s snapshots on 5611. That proves the process's loop is alive on *this* host; it says nothing about whether another host can reach it, because those health sockets bind `127.0.0.1` by default and the image does not publish them. That is one more reason the image `HEALTHCHECK` TCP-probes the engine on 5555 instead: a real connectivity check, just not one that says anything about `pm-index` or any other host.



Every arrow already works with the existing bind/connect pair. What the picture lacks is everything above: nothing authenticates host A to the engine, nothing discovers its address, nothing orchestrates the three hosts as one unit.

12.10 Reference

- `deployment/docker/README.md` — day-to-day usage of the source-built stack
- `deployment/curl/README.md` — the released, pull-only deployment
- Installation — the user-facing view of the same material, including every flag and directory
- Processes, Environment and Ports — what each `pm-*` process does and what the profiles contain
- `docs-design/EduMatcher-Cross-host-connection.md` — an earlier, broader *unimplemented* proposal for cross-host support: per-process `--bind-host` and `--engine-host` flags, a `primary`-IP auto-resolve, port-level environment overrides, and a `network:` section in `engine_config.yaml`. Treat it as the backlog for Part 8, not as current behaviour.

CHAPTER 13

Exchange Experiments Roadmap

This page proposes 20 hands-on experiments that close key gaps between EduMatcher and production exchanges.

Each experiment uses the same format:

- Background
- Financial explanation and motivation (with definitions of all new terms)
- Technical details (scope, outcome, verification)
- Learning objective

Experiments 1-15 are the original proposal, ordered from basic to very complex. Experiments 16-20 were added later (self-trade prevention, plus three further operational and observability tracks) and are appended after, not sorted into that difficulty ordering — see "Suggested progression path" below for where each one fits.

13.1 How to use this page

Statuses verified 2026-09-11

EduMatcher has moved a long way since these experiments were written, and several of them have since been built — sometimes under different names than proposed here. Every `**Status:**` line below was re-checked against the code on that date, and where something shipped, the status says where it lives and how it differs from the proposal. An experiment marked **Implemented** is still worth reading for the financial explanation; it is just no longer something to build.

1. Pick one experiment at a time.
2. Define acceptance criteria before coding.
3. Add tests first when possible.
4. Run the full quality gate after each change.
5. Keep a short implementation log with assumptions and tradeoffs.

13.2 Experiment 1: Price-Time Priority Audit Trace

Status: Still Valid

Difficulty: Basic Estimated effort: Less than 1 day Gap vs real exchange: Real venues provide deterministic post-trade replay and explainability tooling.

Background: Matching is deterministic, but learners often cannot explain why one order filled before another at the same price.

Financial explanation and motivation:

New terms:

- **Queue position**: The rank of a resting order among all orders at the same price on the same side. Orders that arrived earlier have a better (lower) rank and therefore have priority when a matching counterpart arrives.
- **Fill priority**: The sequencing rule applied at match time. Price-time priority means: best price first, and among orders

at the same price, earliest arrival time first.
- **Alpha** (colloquial): A trader's statistical edge — any characteristic that improves their probability of profitable execution. In queue-based markets, a better queue position is itself a form of alpha because it raises fill probability at the target price without changing the price at all.
- **Adverse selection**: The risk that a fill occurs precisely because the counterpart has better information. An order that gets filled by an informed incoming order may find the price moves against the resting side immediately afterward. Understanding queue rank helps reason about when adverse selection is more or less likely.
- **Execution quality**: How closely an actual fill matches the trader's intended price, timing, and quantity. Better queue position tends to improve execution quality because fills happen at the intended price before the market moves.

Queue position is critical in electronic markets because many participants post limit orders at the same price. Being first in queue means your order is matched first when a counterpart arrives. Those at the back of the queue may not be filled at all if the incoming order is smaller than the combined queue depth.

Technical details:

- Add `trace_mode: bool` to engine config. When false, no tracing logic runs and behavior is identical to today.
- At match time, for each candidate resting order evaluated during a single matching cycle, emit a `match.trace` event with fields: `order_id` (str), `price_key` (str decimal), `time_key` (int, monotonic sequence), `side` (str), `was_selected` (bool).
- The selected order emits `was_selected: true`. All other candidates evaluated in the same cycle emit `was_selected: false`.
- Trace events are published to the audit channel before the corresponding `trade.executed` event for the same match.
- The sequence of trace events within one matching cycle must be ordered by `time_key`.

Expected outcome: A student can reconstruct exactly why each fill happened and which queue rule was decisive.

How to verify:

- Unit test: submit two same-price limit buy orders in sequence (order A then order B), then one sell order that fills only order A. Assert A has `was_selected: true` and B has `was_selected: false`.
- Schema test: serialize one `match.trace` event to JSON and lock the exact field names and types in a snapshot test.
- Regression test: run the same scenario with `trace_mode: false` and assert fill events are identical to baseline (no extra events emitted).

Learning objective: Understand deterministic matching and queue priority mechanics at a forensic level.

13.3 Experiment 2: Exchange Tick-Size Regime per Symbol

Status: Partially Implemented — Phase 2 Enhancement

Current state: `tick_decimals` per symbol exists in config. New work: add `tick_size` (exact increment), variable-tick bands, and price validation.

Difficulty: Basic Estimated effort: 1 to 2 days Gap vs real exchange: Production exchanges enforce symbol-specific tick tables.

Background: Current pricing accepts flexible decimal values, but real symbols trade only on valid increments.

Tick concept (new terms):

- A **tick** is the minimum allowed price movement for an instrument. It is the atomic unit of price on that venue for that symbol.
- **Tick size** is the numeric value of that minimum increment (for example `0.01`).
- If tick size is `0.01`, valid prices include `100.00`, `100.01`, `100.02`; invalid examples include `100.005`.
- In general, a price P is valid when `(P - reference_price) mod tick_size == 0`.
- **Bid**: A standing offer to buy at a specified price. The best bid is the highest price any buyer is currently willing to pay.
- **Ask (offer)**: A standing offer to sell at a specified price. The best ask is the lowest price any seller is currently willing to accept.
- **Spread**: The difference between the best ask and best bid. A narrow spread means lower round-trip cost for traders who need to both buy and sell.
- **Depth**: The quantity available at each price level in the order book, not just the best bid and ask.
- **Market maker**: A participant who continuously posts both a bid and an ask, earning the spread as compensation for providing liquidity. Market makers are sensitive to tick size because it defines the minimum spread they can quote.

Why exchanges enforce this:

- It standardizes price competition and prevents fractional undercutting noise.
- It improves queue fairness by forcing participants to compete in meaningful increments.
- It directly shapes spread and liquidity depth.

Financial explanation and motivation: Tick size is a deliberate market design parameter. Larger tick sizes widen spreads and concentrate depth at fewer price levels, making queue position more valuable. Smaller tick sizes tighten spreads but fragment liquidity across more levels. Market makers, latency traders, and execution algorithms all adapt their behavior when tick regimes change.

Technical details:

- Extend symbol config with `tick_size: Decimal` (required, no default) and optional `tick_bands: list[{from_price, to_price, tick_size}]` for variable-tick regimes.
- Validate price in the NEW order path, AMEND path, stop-limit trigger price, and iceberg limit price, all using the same `is_valid_tick(price, symbol_config)` function.
- Reject invalid prices with reason code `INVALID_TICK` and include `{price, tick_size, symbol}` in the reject payload.
- Implement `is_valid_tick` using integer arithmetic: multiply both `price` and `tick_size` by a common scale factor to make them integers, then check divisibility. Do not use floating-point modulo.

Expected outcome: Invalid price steps are blocked; valid prices align to venue rules.

How to verify:

- Parametric tests: for each of three symbols with different tick sizes (`0.01`, `0.25`, `0.50`), test at least one valid and two invalid price values on NEW and AMEND paths.
- Test stop-limit trigger price and iceberg limit price separately.
- Verify reject payload contains `symbol`, `price`, and `tick_size`.
- Verify existing valid orders on all paths are unaffected.

Learning objective: See how market design constraints shape order entry and liquidity formation.

13.4 Experiment 3: Session-Aware Order Entry Policy Matrix

Status: Partially Implemented — Phase 2 Enhancement

Current state: Session states (PRE_OPEN, OPENING_AUCTION, CONTINUOUS, CLOSING_AUCTION, CLOSED) and phase checking exist. New work: add per-phase admissibility matrix for (order_type, TIF) combinations.

Difficulty: Basic Estimated effort: 1 to 2 days Gap vs real exchange: Venues apply strict policy by phase (pre-open, auction, continuous, close).

Background: The scheduler changes phases, but policy granularity can be expanded to reflect real venue behavior.

Financial explanation and motivation:

New terms:

- **Session phase**: A distinct period of the trading day with different rules for what can be submitted and matched. The five standard phases are:
  - `pre-open`: orders may be entered but no matching occurs; the book is being populated.
  - `auction`: orders are collected for a batch match; continuous matching is suspended.
  - `continuous`: normal order-by-order matching, the most common intra-day state.
  - `close`: wind-down period with restricted entry; typically only limit orders are accepted.
  - `halt`: all activity is frozen, usually due to a news event or circuit breaker.
- **TIF (Time-In-Force)**: The policy controlling how long an order remains active. Common values across exchanges: `DAY` (expires at session end), `GTC` (Good Till Cancelled, survives across sessions), `IOC` (Immediate-or-Cancel: fill what you can right now, cancel any unfilled remainder), `FOK` (Fill-or-Kill: fill the entire quantity immediately or cancel entirely). In EduMatcher specifically, `IOC` and `FOK` are `OrderType` values, not `TIF` values — the real `TIF` enum has only `DAY/GTC/ATO/ATC` (see the Reference Manual glossary). The matrix below follows that split: `IOC` gets its own column rather than pairing with a `TIF`.
- **Order admissibility**: Whether a given order type, optionally with a TIF, is permitted during a given phase. For example, Market orders are blocked during pre-open because no reference price exists for a fair execution.

- **Opening volatility**: The price instability at the start of the continuous session, caused by accumulated overnight orders. Session policies reduce this risk by restricting the most unpredictable order types until the auction establishes an opening price.

A session policy matrix defines admissibility as a two-dimensional table: rows are session phases, columns are order types (with a TIF where the order type takes one), and cells are ALLOW or DENY. Real exchanges publish these tables in their rulebooks.

Example partial matrix:

Phase	Market/DAY	Limit/DAY	Limit/GTC	IOC
pre-open	DENY	ALLOW	ALLOW	DENY
auction	DENY	ALLOW	ALLOW	DENY
continuous	ALLOW	ALLOW	ALLOW	ALLOW
close	DENY	ALLOW	DENY	DENY
halt	DENY	DENY	DENY	DENY

Technical details:

- Define a `session_policy` config section: a mapping from phase name to a list of allowed `(order_type, tif)` pairs. Anything not listed is denied (allowlist approach).
- Enforce in engine validation before any book processing or risk checks.
- Reject with reason code `PHASE_POLICY_VIOLATION` and include `{phase, order_type, tif}` in the reject payload.
- Policy is evaluated at each phase transition without restarting the engine.

Expected outcome: Different order classes are accepted or rejected according to phase; policy violations produce informative rejects.

How to verify:

- Transition programmatically through all five phases in tests.
- For each phase, test at least one ALLOW and one DENY combination from the matrix.
- Verify reject payload contains `phase`, `order_type`, and `tif`.
- Verify existing passing combinations in `continuous` phase are unaffected.

Learning objective: Understand how session design controls market behavior and participant risk.

13.5 Experiment 4: Per-Gateway Rate Limits and Burst Controls

Status: Implemented (gateway layer) — verified 2026-09-11

Rate limiting is enforced at the gateways rather than in the engine: `alf_gwy/gateway.py` and `balf_gwy/gateway.py` reject with `RATE_LIMITED`, and `api_gateway` maps its own `RATE_LIMIT` to the same code. What remains open from the proposal below is the *burst* shaping (token bucket with a configurable refill) and per-gateway limits in config rather than in code.

Difficulty: Intermediate Estimated effort: 1 to 2 days Gap vs real exchange: Real gateways enforce throughput and abuse controls.

Background: Bot and human gateways can currently submit at very high rates without any throttle.

Financial explanation and motivation:

New terms:

- **Throughput**: The number of messages (orders, amends, cancels) per unit time that a gateway is permitted to send to the exchange.
- **Burst**: A short-duration spike in submission rate above the sustained throughput limit. Real exchanges allow moderate bursts to accommodate legitimate clustering of events (e.g., a strategy responding to news).
- **Token bucket**: A rate-limiting algorithm. A bucket holds up to `capacity` tokens. Tokens refill at `refill_rate` tokens per `refill_interval_ms`. Each incoming message consumes one token. If the bucket is empty, the message is rejected

or queued depending on policy. This allows bursts up to `capacity` while limiting the sustained rate to `refill_rate / refill_interval_ms` messages per millisecond.
- **Queue stuffing**: A manipulative tactic where a participant floods the exchange with orders and cancels in rapid succession with no real intent to trade, in order to slow down competitor systems and exchange processing. Prohibited by most regulators.
- **Fair access**: The principle that all market participants have equal opportunity to submit orders within their entitlements, without one participant's volume degrading the service seen by others.

Rate limits protect fair access and market stability. Without them, a single misbehaving participant — or a bug in a trading algorithm — can degrade exchange performance for all others.

Technical details:

- Add a `rate_limit` config section per gateway with fields: `capacity` (int, maximum burst tokens), `refill_rate` (int, tokens added per interval), `refill_interval_ms` (int), `policy` (str: `reject` or `delay`).
- Track NEW, AMEND, and CANCEL message types against a single shared bucket per gateway (configurable to separate buckets per type).
- On empty bucket: `reject` policy emits a `RATE_LIMIT_EXCEEDED` reject immediately; `delay` policy queues the message until a token is available, up to a `max_queue_depth`; overflow beyond that is rejected.
- Expose per-gateway counters via the stats channel: `tokens_remaining`, `messages_accepted`, `messages_rejected`, `messages_delayed`
- Bucket state resets at each session start.

Expected outcome: Gateway behavior is constrained to configured throughput with observability; under-limit flow is unaffected.

How to verify:

- Stress test: submit 2× the bucket `capacity` in a single burst. Assert exactly `capacity` messages are accepted and the rest are rejected (reject policy).
- Verify counter values match accepted and rejected counts exactly.
- Verify under-limit flow (sending at the refill rate) produces zero rejections over a sustained window.
- Refill test: exhaust the bucket, wait one `refill_interval_ms`, confirm exactly one more message is accepted.

Learning objective: Learn the tradeoff between open access, fairness, and operational safety.

13.6 Experiment 5: Fee and Rebate Model (Maker-Taker)

Status: Still Valid

Difficulty: Intermediate Estimated effort: 1 to 2 days Gap vs real exchange: Real venues expose fee schedules that alter trader behavior.

Background: Clearing currently tracks profit and loss without exchange fee microstructure.

Financial explanation and motivation:

New terms:

- **PnL (Profit and Loss)**: The net financial result of trading activity. **Gross PnL** is computed solely from fill prices and quantities (buy-sell price difference times quantity). **Net PnL** is gross PnL minus all fees paid plus all rebates received.
- **Liquidity**: The ease with which a participant can buy or sell without significantly moving the price. High liquidity means tight spreads and large available quantities.
- **Maker**: An order that *adds* liquidity to the book. A limit order that does not immediately match rests passively; when it is later filled by an incoming order, it is the maker side of that trade.
- **Taker**: An order that *removes* liquidity from the book. Any order that matches immediately — market orders, or limit orders aggressive enough to cross the spread — is the taker side.
- **Maker rebate**: A payment made by the exchange *to* the maker. The exchange pays this rebate to incentivize participants to post resting orders and provide continuous liquidity.
- **Taker fee**: A charge levied by the exchange *on* the taker. It is the primary revenue source for exchanges using the maker-taker model. The taker fee is typically larger than the maker rebate, and the difference is the exchange's margin.
- **Notional value**: The total monetary value of a trade: `price × quantity`. Fee rates are applied to this figure.
- **Fee schedule**: The table of fee and rebate rates, typically expressed as a fraction of notional value. For example, a

taker rate of `0.0030` means 0.30% of trade notional.

Example calculation: taker rate `0.0030`, maker rate `-0.0010` (negative = rebate). Fill of 100 shares at $50.00 (notional $5,000):

- Taker pays: $5,000 × 0.0030 = **$15.00 fee**
- Maker receives: $5,000 × 0.0010 = **$5.00 rebate** (a credit, not a charge)

Fee structures change strategy incentives. A maker rebate encourages posting passive limit orders. A taker fee discourages aggressive flow unless the price signal is strong enough to justify the cost.

Technical details:

- Add `fee_schedule` section to symbol config with fields: `taker_rate: Decimal`, `maker_rate: Decimal`. A negative `maker_rate` means a rebate (credit to the participant).
- Compute fees at fill time in clearing. For each fill event: `fee = price * qty * rate`, where rate is `taker_rate` for the aggressor side and `maker_rate` for the resting side.
- Extend the clearing fill record and session report with: `gross_pnl`, `fee_charged` (positive = paid by participant), `rebate_received`, `net_pnl`.
- Fees accumulate per participant per session.
- When `fee_schedule` is absent for a symbol, behavior is identical to today (zero fees, backward compatible).

Expected outcome: Net PnL differs from gross PnL when fees are active; fee-free mode remains backward compatible.

How to verify:

- Controlled scenario: one maker and one taker, known fill price and quantity. Assert `fee_charged` and `rebate_received` match hand calculations to full decimal precision.
- Verify maker-side rebate produces a negative `fee_charged` (a credit).
- Verify taker-side fee increases total cost above the raw execution price.
- Run with no `fee_schedule` and assert all fee fields are zero and gross PnL is unchanged.

Learning objective: Understand execution economics beyond simple price and quantity.

13.7 Experiment 6: Realistic Cancel/Amend Latency Simulation

Status: Still Valid

Difficulty: Intermediate Estimated effort: 1 to 2 days Gap vs real exchange: Matching systems have non-zero network and processing latency.

Background: Immediate local execution can mask race conditions that dominate live markets.

Financial explanation and motivation:

New terms:

- **Latency**: The time delay between sending a message (order, cancel, amend) and the exchange processing it. In production this is measured in microseconds to low milliseconds.
- **Cancel latency risk**: The risk that a cancel request arrives at the engine *after* the order has already been filled, because a matching counterpart arrived in the same narrow time window. The cancel "loses the race" and the fill stands.
- **Quote fade**: When a passive resting order is cancelled just before an incoming aggressive order would have matched it. From the aggressor's perspective, the liquidity appeared to be there but vanished. Quote fade is a significant friction in fast markets.
- **Race condition**: A situation where the final outcome depends on the relative timing of two or more independent events. Cancel-vs-fill races are a major source of operational risk in live trading systems.
- **Adverse selection (HFT context)**: When a fill occurs specifically because an informed participant's order arrived faster than the resting side's cancel. The resting participant is "picked off" — filled at a price that immediately moves against them.

In a local simulation, all messages are processed in strict submission order with zero delay. This hides behavioral differences that are important in production: a cancel submitted 200µs before an aggressive order might arrive *after* it if the aggressive order takes a faster network path.

Technical details:

- Add optional `latency_simulation` config section with fields: `enabled: bool`, `distribution: str` (one of `uniform`, `normal`, `lognormal`), `min_ms: float`, `max_ms: float`, `mean_ms: float`, `stddev_ms: float`, `seed: int`.
- Implement as a message-queue wrapper: each incoming message is timestamped on arrival and held for a delay drawn from the configured distribution. The internal queue processes messages in timestamp-order (earliest delivery time first).
- Provide three named presets selectable by `profile` string: `low` (uniform 0.1–1ms), `medium` (uniform 1–20ms), `stressed` (uniform 10–200ms). Individual field values override presets.
- Use `seed` for fully reproducible replays: the same seed always produces the same delay sequence.
- When `enabled: false`, all messages are processed in submission order as today, with no timing overhead.

Expected outcome: Race conditions become observable and repeatable; strategy robustness can be measured under different latency profiles.

How to verify:

- Seeded determinism test: run the same scenario twice with the same seed and assert fill outcomes are identical.
- Bounds test: over 10,000 draws, confirm no delay falls outside `[min_ms, max_ms]`.
- Race test: submit a cancel and an aggressive order in sequence; under non-zero latency the race outcome may differ from zero-latency. Assert each mode produces consistent deterministic output.
- Disabled test: run the same scenario with `enabled: false` and assert fill events are byte-identical to the baseline.

Learning objective: See how latency changes strategy outcomes and risk exposure.

13.8 Experiment 7: Full Order State Machine with Reason Codes

Status: Partially Implemented — Phase 2 Enhancement

Current state: Order status enum (NEW, PARTIAL, FILLED, CANCELLED, REJECTED, EXPIRED) exists. New work: formalize state transition table, validate transitions, add machine-readable reason codes on all lifecycle events.

Difficulty: Intermediate Estimated effort: 1 to 3 days Gap vs real exchange: Real systems standardize lifecycle states and reject codes.

Background: Order status is present, but lifecycle transitions can be made more explicit, exhaustive, and auditable.

Financial explanation and motivation:

New terms:

- **Order state machine**: A formal description of all states an order can be in and all permitted transitions between them. A state machine prevents illegal combinations such as amending a fully-filled order.
- **Terminal state**: A state from which no further transitions are possible. `FILLED`, `CANCELLED`, `EXPIRED`, and `REJECTED` are all terminal. Once an order reaches a terminal state it cannot be modified.
- **Reason code**: A machine-readable string attached to terminal or exceptional transitions, explaining why the transition occurred. Examples: `CANCELLED_BY_USER`, `EXPIRED_SESSION_END`, `REJECTED_INVALID_TICK`. Reason codes allow automated systems to handle each case correctly without parsing free-text messages.
- **Partial fill**: A state where an order has been matched for less than its full quantity and continues to rest in the book for the remainder. The partially-filled state is not terminal.
- **Reconciliation**: The process of comparing two independent records to verify they agree. An operations team reconciles their internal order records against the exchange's records to confirm every order reached a known terminal state and no fills are missing. Stable reason codes make this comparison unambiguous.

Valid state progression (all paths must terminate):

```
NEW → ACKNOWLEDGED → ACTIVE → FILLED (terminal)
                   → PARTIALLY_FILLED → FILLED (terminal)
                   → PARTIALLY_FILLED → CANCELLED (terminal, filled-qty preserved)
                   → CANCELLED (terminal)
                   → EXPIRED (terminal)
NEW → REJECTED (terminal)
```

Illegal transitions that must be blocked: amend on any terminal order, cancel on a rejected order, fill on a cancelled order.

Technical details:

- Introduce an `OrderState` enum containing all states listed above.

- Add a `transition(order, new_state, reason_code)` function that validates the proposed transition against an allowable-transitions table. Raise `InvalidStateTransition` if the transition is not permitted.
- All engine handlers that mutate order state must call `transition()` rather than setting state directly.
- Emit reason codes in all state-change events: ACK, FILL, CANCEL, EXPIRE, REJECT.
- Reason code format: `SCREAMING_SNAKE_CASE`, max 64 characters, stable across versions (never rename a code that has shipped).

Expected outcome: Orders always follow valid state paths; all terminal transitions carry a machine-parseable reason code.

How to verify:

- Transition matrix test: for every (current_state, new_state) pair, assert `transition()` either succeeds or raises `InvalidStateTra` as expected per the table.
- Integration tests across: amend (active → active), partial fill, cancel after partial fill, session expiry.
- Snapshot test: lock all reason code strings so accidental renames are caught in CI.
- Illegal transition test: attempt to fill a cancelled order and assert `InvalidStateTransition` is raised.

Learning objective: Understand order lifecycle reliability and protocol discipline.

13.9 Experiment 8: Market Data Channel Separation (Level 1 vs Level 2)

Status: Partially Implemented — Phase 2 Enhancement

Current state: Depth metrics via `depth_snapshot()` and `encode(f"depth.{symbol}", depth)` publication exist. New work: introduce formal `book.l1.*` and `book.l2.*` topic namespaces with contract-enforced payloads, configurable publish cadence, and sequence counters per channel.

Difficulty: Intermediate Estimated effort: 2 to 3 days Gap vs real exchange: Production venues separate feed products and entitlement tiers.

Background: Current subscribers consume broad book topics with a single payload format, receiving more data than many consumers need.

Financial explanation and motivation:

New terms:

- **Level 1 data** (top-of-book): The minimum market data product. It contains only four values per symbol: best bid price, best bid quantity, best ask price, and best ask quantity. Sufficient for most simple strategies and retail display.
- **Level 2 data** (market depth): The full order book at all available price levels. Each entry lists a price level, the total quantity of all resting orders at that price, and the count of those orders. Essential for market makers and execution algorithms that need to see where liquidity is concentrated.
- **Top-of-book**: The single best price on each side (highest bid, lowest ask). Level 1 carries only this.
- **Depth**: All price levels beyond the best bid and ask, showing available liquidity at each increment.
- **Entitlement tier**: A subscription category that controls which data products a participant receives. Level 1 is cheap and widely distributed; Level 2 is more expensive and used by professionals. Separate channels allow the exchange to enforce and monetize this distinction.
- **Strategy edge from depth data**: Knowing that a large resting order sits three ticks below the best bid can inform execution decisions — it signals a likely price support level. This information is not available from Level 1 alone.

Technical details:

- Define two topic namespaces: `book.l1.<symbol>` and `book.l2.<symbol>`.
- Level 1 payload schema (7 fields): `{symbol, seq, ts, bid_price, bid_qty, ask_price, ask_qty}`.
- Level 2 payload schema: `{symbol, seq, ts, bids: [{price, qty, count}], asks: [{price, qty, count}]}`. Bids ordered descending by price (best first); asks ordered ascending by price (best first).
- Add `publish_cadence_ms` config per channel: `l1` defaults to publishing on every book change; `l2` defaults to 100ms throttled snapshots.
- Each channel maintains its own sequence counter, starting at 1 each session.
- The existing `book.*` topic continues to function during this transition (deprecated but not removed).

Expected outcome: Consumers can subscribe to exactly the data product they need; payload size difference between L1 and L2 is measurable.

How to verify:

- Contract test for `book.l1.*`: assert the payload contains exactly the 7 L1 fields and no depth arrays.
- Contract test for `book.l2.*`: assert `bids` and `asks` arrays are present, non-empty, and each array is correctly ordered.
- Size test: run a 100-order scenario and measure the average L1 vs L2 payload byte size.
- Backward compatibility: confirm existing `book.*` subscribers still receive data unchanged.

Learning objective: Understand market data product design and distribution tradeoffs.

13.10 Experiment 9: User and Account Domain Refactor

Status: Still Valid

Difficulty: Complex Estimated effort: 3 to 5 days Gap vs real exchange: Venues operate with firm, user, account, and permission hierarchies.

Background: Gateway ID currently acts as both identity and account boundary. Real exchanges separate these concerns.

Financial explanation and motivation:

New terms:

- **Firm**: The legal entity — broker, market maker, or proprietary trading firm — that has a contractual relationship with the exchange. A firm has one or more trading accounts.
- **Account**: A trading account owned by a firm. Risk limits, positions, fills, and PnL are all tracked at the account level. A firm may have multiple accounts for different desks, strategies, or regulatory purposes.
- **Gateway (connection)**: The network session between a trading system and the exchange. A single account may have multiple gateways for redundancy (so that a network failure on one connection does not halt trading). Multiple accounts at the same firm may also share a single gateway connection.
- **Risk boundary**: The unit at which position limits, notional limits, and loss limits are enforced. Real exchanges enforce at the account level, not the connection level. This means limits apply consistently regardless of which gateway submits the order.
- **Compliance**: The obligation to meet regulatory and exchange rules. Account-level tracking enables the audit trails required by regulators: which account placed which order, when, and for what stated purpose.

The current model conflates connection identity with trading account. This breaks down when a firm runs redundant gateways (both map to the same account and positions should aggregate), or when one gateway needs to manage orders across multiple accounts.

Technical details:

- Introduce an `Account` domain object: `{account_id: str, firm_id: str, gateway_ids: list[str]}`.
- All order messages include `account_id` in addition to `gateway_id`.
- Position tracking, fill records, and PnL reports are keyed by `account_id`, not `gateway_id`.
- Cancel-on-disconnect cancels all open orders whose `account_id` maps to the disconnecting gateway, but only those submitted by that specific gateway (not orders submitted by other gateways on the same account, unless configured differently).
- Authorization: a gateway may only act on `account_id` values listed in its own registration.
- Migration path: if `account_id` is absent in an incoming message, derive it from `gateway_id` using a fallback lookup table (backward compatible with existing configs).

Expected outcome: Multiple gateways can share one account and positions aggregate correctly; controls apply consistently at the account level.

How to verify:

- Multi-gateway test: two gateways submit orders for the same `account_id`. Assert positions sum at the account level.
- Cancel-on-disconnect test: disconnect one gateway, assert only its orders are cancelled (not orders from the other gateway on the same account).
- Authorization test: a gateway that attempts to submit for an `account_id` it is not registered to is rejected.
- Migration test: a message without `account_id` is accepted and the correct account is derived via fallback.

Learning objective: Understand how real exchange participants are modeled beyond a single connection ID.

13.11 Experiment 10: Kill Switch and Fat-Finger Controls

Status: Largely Implemented — verified 2026-09-11

The kill switch shipped with three scopes, not the two proposed here: `risk.kill_switch` (per symbol), `risk.kill_switch_gateway`, and `risk.kill_switch_global`, each with a `command_id` that the resulting `order.cancelled` events carry so a post-mortem can join cancels back to the command that caused them. Fat-finger limits exist as `MAX_ORDER_QTY`, `MAX_ORDER_VALUE` and `COLLAR_BREACH` reject codes — the names differ from the `FAT_FINGER_*` ones proposed below, and the collar check subsumes `max_price_distance_ticks`. Still open: per-*account* scope (EduMatcher has no account domain — see Experiment 9) and a separate privileged auth token for the kill message.

Current state: Kill-switch gateway cancellation exists (`_handle_kill_switch()`). New work: hard-stop governance (privileged-only reset), notional value guards (`max_order_notional`), and price-distance guardrails (`max_price_distance_ticks`).

Difficulty: Complex Estimated effort: 2 to 4 days Gap vs real exchange: Exchanges and brokers provide hard kill-switch controls.

Background: Runaway order flow needs immediate containment beyond normal strategy logic.

Financial explanation and motivation:

New terms:

- **Fat-finger error**: An accidental order entry mistake — for example, submitting 1,000,000 shares instead of 1,000, or a price of $0.01 instead of $100.01. Fat-finger events have caused multi-million dollar losses and temporary market disruptions on real venues.
- **Notional value**: The total monetary value of an order: `price × quantity`. A fat-finger check compares this against a configured maximum to catch orders that are impossibly large relative to normal activity.
- **Price distance guardrail**: A check that rejects any order whose price is more than a configured number of ticks away from the current best price (or last trade price). This prevents orders accidentally submitted at the wrong end of the price scale from resting in the book.
- **Kill switch**: An emergency mechanism — triggered by a risk manager or automated monitor — that immediately cancels all open orders for a gateway or account and blocks all further new orders until re-enabled by a privileged source.
- **Hard stop**: A kill-switch state that cannot be reset by the gateway itself. Only a separate privileged process (the risk manager endpoint) can re-enable the gateway. This prevents a buggy algorithm from resetting its own kill switch.

Technical details:

- Add kill-switch state per gateway and per account: `active` (normal) or `killed`.
- Kill switch API: a privileged `KILL` message (distinct from order messages, requires a separate auth token) transitions state to `killed`, triggers immediate cancel-all for all open orders of the target, and blocks all NEW and AMEND messages. Only a `RESET` message from the same privileged source can return to `active`.
- Add `max_order_notional: Decimal` per gateway config. Reject any NEW or AMEND where `price * qty > max_order_notional` with reason code `FAT_FINGER_NOTIONAL`.
- Add `max_price_distance_ticks: int` per symbol config. Reject any order priced more than this many ticks from the last published best price on the same side. Reason code: `FAT_FINGER_PRICE_DISTANCE`.
- Emit a `KILL_SWITCH_TRIGGERED` audit event with: timestamp, target (gateway or account ID), trigger source, and count of orders cancelled.

Expected outcome: Unsafe flow is stopped instantly and deterministically; every kill and reset action produces a clear audit trail.

How to verify:

- Runaway scenario: submit 50 orders from one gateway, trigger kill switch, assert all 50 open orders are cancelled within one processing cycle and no further orders are accepted.
- Notional test: submit an order with `price * qty > max_order_notional` and assert `FAT_FINGER_NOTIONAL` reject with the correct payload fields.
- Price distance test: submit an order 100 ticks from the best price and assert `FAT_FINGER_PRICE_DISTANCE` reject.
- Hard stop test: confirm the gateway cannot self-reset; confirm a privileged `RESET` message re-enables it.
- Audit test: assert the `KILL_SWITCH_TRIGGERED` event exists with the correct cancelled-order count.

Learning objective: Learn real-time safety controls used in production trading systems.

13.12 Experiment 11: Drop Copy and Post-Trade Reconciliation Stream

Status: Largely Implemented — verified 2026-09-11

Drop copy shipped as `drop_copy.event.{gateway_id}` with a session-global `seq`, plus `drop_copy.replay.{recipient_id}` for gap recovery — a live replay channel rather than the proposed end-of-session log file. Reconciliation is `pm-clearing-cli reconcile` rather than a standalone `dropcopy_reconcile.py`. Still open from the proposal: the durable `dropcopy_YYYYMMDD.log` JSON-lines artifact, if you want reconciliation against a file rather than against the clearing database.

Current state: Drop copy publisher, sequencing, and in-memory replay buffer exist (`DropCopyPublisher`). New work: durable append-only log file, session-scoped rotation, reconciliation tooling for gap detection and quantity validation.

Difficulty: Complex Estimated effort: 2 to 4 days Gap vs real exchange: Participants usually consume independent drop copy for control and compliance.

Background: Execution feedback currently arrives only through strategy-facing channels. There is no independent audit stream.

Financial explanation and motivation:

New terms:

- **Drop copy**: An independent, real-time copy of all order lifecycle events (new, acknowledged, filled, cancelled, expired) sent to a separate consumer. The name comes from the physical practice of "dropping" a carbon copy of each trade ticket for the operations desk. Drop copy is consumed by risk managers, compliance teams, and operations staff who monitor activity without being part of the trading strategy itself.
- **Sequence number**: A monotonically increasing integer assigned to every drop-copy event in order. Recipients use sequence numbers to detect gaps — if event 1042 arrives after 1040, the recipient knows event 1041 is missing and must be requested.
- **Gap detection**: The process of identifying missing events in a numbered sequence. A sequence gap means a fill, cancel, or rejection may have been silently lost, creating a discrepancy between the exchange's record and the participant's record.
- **Reconciliation** (post-trade): The end-of-day (or intra-day) process of comparing a participant's internal trade records against the exchange's official drop-copy records to find any discrepancies. Unresolved discrepancies must be escalated.
- **Durable replay**: The ability to re-read past events from a stored log, in order, to recover state after a crash or fill in a detected gap.

Technical details:

- Add a `dropcopy.*` topic namespace mirroring all order lifecycle events: `dropcopy.new`, `dropcopy.ack`, `dropcopy.fill`, `dropcopy.cancel`, `dropcopy.expire`, `dropcopy.reject`.
- Each drop-copy event includes: `seq` (int, monotonically increasing per gateway, starting at 1), `ts` (microsecond timestamp), and the full original event payload unchanged.
- Write all drop-copy events to a durable append-only log file `dropcopy_YYYYMMDD.log` in JSON-lines format (one event per line). The file is closed and renamed at session end.
- Provide a `dropcopy_reconcile.py` utility that reads the log file and a participant's local fill records and outputs three lists: missing events (gap in sequence), duplicate events (seq appears twice), quantity mismatches (fill qty in log differs from local record).
- Sequence numbers reset to 1 at each session start.

Expected outcome: Operations can verify completeness and detect missing events; the reconciliation utility identifies all gaps in a controlled scenario.

How to verify:

- Submit 100 orders through their full lifecycle. Assert the drop-copy log file contains exactly the expected number of events with no sequence gaps.
- Gap detection test: delete one line from the log file, run reconciliation, assert it reports exactly one gap at the correct sequence number.
- End-to-end replay: feed the drop-copy log to a stateless consumer and assert it reconstructs the correct final position for each account.
- Quantity mismatch test: modify one fill qty in the log, assert reconciliation flags it.

Learning objective: Understand operational controls and post-trade integrity pipelines.

13.13 Experiment 12: Matching Mode Plug-in Layer (Pro-Rata vs Price-Time)

Status: Still Valid

Difficulty: Complex Estimated effort: 4 to 7 days Gap vs real exchange: Different asset classes and venues use different allocation algorithms.

Background: The current engine uses a single matching priority (price-time). Many real venues — particularly derivatives and fixed income — use alternative algorithms.

Financial explanation and motivation:

New terms:

- **Price-time priority**: The standard equity matching rule. Among all orders at the same price, the one that arrived first is filled first. Queue position (time of entry) is the decisive factor after price.
- **Pro-rata allocation**: An alternative matching rule common in derivatives markets. When a new order matches at a price where multiple resting orders exist, the incoming quantity is distributed *proportionally* to the resting size of each order at that price, regardless of arrival time.
  - Example: three resting orders at the same price with sizes 100, 200, 300 (total 600). An incoming order for 300 arrives. Allocations: $100 \times (300/600) = 50$, $200 \times (300/600) = 100$, $300 \times (300/600) = 150$.
  - Remainders from integer rounding are distributed to the largest under-allocated order(s) using price-time order as a tiebreak.
- **Queue value**: Under price-time, being first in queue is extremely valuable. Under pro-rata, queue position does not matter — size is everything. This fundamentally changes who provides liquidity and how they manage resting orders.
- **Fill fragmentation**: The average number of separate fill events that a single incoming order generates. Price-time typically produces low fragmentation (starts with the first resting order). Pro-rata produces high fragmentation (splits across many resting orders). High fragmentation increases post-trade processing cost.
- **Market maker economics under pro-rata**: Market makers are incentivized to post large resting sizes (to receive a larger proportional share of each fill). This tends to increase displayed liquidity but can also inflate book size without proportional depth improvement.

Technical details:

- Define a `MatchAllocator` protocol with a single method: `allocate(incoming_qty: int, candidates: list[RestingOrder]) -> list[Allocation]`.
- Implement `PriceTimeAllocator` (refactored existing behavior) and `ProRataAllocator` (new).
- Pro-rata implementation: `allocation_i = floor(incoming_qty * size_i / total_resting_size)`. Distribute any unallocated remainder quantity one unit at a time to the largest fractional-remainder orders, using time_key to break ties.
- Both allocators must produce fills that sum exactly to `min(incoming_qty, total_resting_qty)`. Add an assertion for this invariant.
- Configure per symbol via `matching_mode: price_time | pro_rata` in symbol config. Default is `price_time`.

Expected outcome: The same order flow yields measurably different fills under the two allocation policies.

How to verify:

- Price-time scenario: three resting orders at the same price. Assert first-arrived is filled first.
- Pro-rata scenario: same three orders. Assert fills are proportional to size with correct remainder handling.
- Overflow scenario: incoming order larger than all resting liquidity. Assert all resting orders are fully filled under both modes.
- Fragmentation measurement: run a 1,000-order scenario under both modes and compare average fills-per-incoming-order.
- Backward compatibility: run the full regression suite with `matching_mode: price_time` (default) and assert no behavior changes.

Learning objective: See how market design choices alter participant incentives and microstructure dynamics.

13.14 Experiment 13: Cross-Product Risk and Portfolio Margin Approximation

Status: Still Valid

Difficulty: Very Complex Estimated effort: 1 to 2 weeks Gap vs real exchange: Risk is usually portfolio-aware across correlated products.

Background: Current controls are per-symbol and per-gateway. No cross-product risk netting is applied.

Financial explanation and motivation:

New terms:

- **Position**: The net quantity of a symbol held by an account. A **long position** means more bought than sold (the account gains if price rises). A **short position** means more sold than bought (the account gains if price falls).
- **Portfolio**: The complete set of positions across all symbols held by an account.
- **Margin**: A capital deposit or requirement an account must maintain to support its open positions. It acts as a buffer against potential losses. If the account's losses would exceed the margin, the exchange may force-close positions.
- **Portfolio margin** (risk-based margin): A margin calculation that accounts for correlations between positions. If an account is simultaneously long Symbol A and short Symbol B, and A and B tend to move together, the combined risk is lower than the sum of individual risks. Portfolio margin rewards this hedging by requiring less total margin, freeing capital for other trades.
- **Correlation**: A number between -1.0 and +1.0 expressing how closely two instruments move together. Correlation +1.0: identical movement. Correlation -1.0: opposite movement. Correlation 0: no relationship.
- **Haircut**: A reduction applied to the offset credit granted for a correlated hedge. A correlation of 0.85 with a haircut factor of 0.80 means the hedge only offsets `0.85 × 0.80 = 68%` of the opposing position's margin requirement.
- **Mark-to-market (MtM)**: Revaluing all positions at the current market price to compute unrealized PnL and current margin requirements. Done continuously in production.
- **Stress test**: Applying a hypothetical extreme price move (for example, all prices fall 10% simultaneously) to current positions to verify that margin requirements would cover the resulting loss.

Example: Account holds +100 AAPL (long) and −80 MSFT (short). AAPL/MSFT correlation is 0.85. Siloed margin charges full margin for each independently. Portfolio margin reduces the total requirement because the short MSFT offsets some of the AAPL long-side risk.

Technical details:

- Add a `RiskEngine` module that subscribes to fill events and maintains per-account positions in memory, recomputing after each fill.
- Load a symbol correlation matrix from config: `correlations: {AAPL: {MSFT: 0.85}, MSFT: {AAPL: 0.85}}`.
- Per-symbol margin rate configured as `symbol_margin_rate: Decimal` per symbol.
- Portfolio margin formula: compute base margin as `sum(abs(position_i) * price_i * symbol_margin_rate_i)` for all symbols. Then for each pair of offsetting positions (one long, one short) with positive correlation, apply a haircut credit: `credit = min(margin_A, margin_B) * correlation_AB * haircut_factor`. Net margin = base margin − sum(all credits).
- If portfolio margin exceeds `margin_limit` for the account, reject any new order that would increase net exposure, with reason `MARGIN_LIMIT_EXCEEDED`.
- Emit `risk.margin_update` event after each recomputation: `{account_id, portfolio_margin, margin_limit, headroom}`.

Expected outcome: Risk controls become portfolio-sensitive; accounts with offsetting positions can hold larger combined positions than siloed limits would allow.

How to verify:

- Scenario A: two uncorrelated symbols (correlation 0.0). Assert portfolio margin equals the sum of individual margins.
- Scenario B: two correlated symbols (correlation 0.90, haircut 0.80). Assert portfolio margin is lower than the sum by the expected haircut amount, verified by hand calculation.
- Scenario C: submit a risk-increasing order that pushes portfolio margin above `margin_limit`. Assert `MARGIN_LIMIT_EXCEEDED` reject.
- Stress test: apply a 10% price shock to all symbols and verify portfolio margin increases proportionally.

Learning objective: Understand portfolio-level risk governance in modern exchanges and clearing ecosystems.

13.15 Experiment 14: Auction Imbalance Feed and Indicative Price Model

Status: Implemented — verified 2026-09-11

`auction.indicative.{symbol}` is published during call phases and `auction.result.{symbol}` at the uncross. The field names differ from the proposal below: the spec uses `eq_price/eq_qty` for the indicative equilibrium, and a single `imbalance_side` + `imbalance_qty` pair rather than separate `buy_imbalance_qty/sell_imbalance_qty`. Read `spec/messages/auction.yaml` for the authoritative shape; the financial explanation below is still the best description of *why* the feed exists.

Difficulty: Very Complex Estimated effort: 1 to 2 weeks Gap vs real exchange: Opening and closing auctions publish rich imbalance signals.

Background: Session support exists, but indicative auction data is not yet computed or published.

Financial explanation and motivation:

New terms:

- **Auction**: A trading session phase where all buy and sell orders are collected over a window and then matched at a single clearing price when the window closes. Unlike continuous trading, no matching occurs while the auction is collecting orders.
- **Uncross**: The moment when the auction ends and the engine matches all compatible orders. Every eligible buy order at or above the clearing price and every eligible sell order at or below it is filled at that single price simultaneously.
- **Auction clearing price**: The price that maximizes the total quantity matched. This is found by scanning all resting price levels and computing, for each candidate price P, how many units can be matched: `matched(P) = min(total_buy_qty_at_or`
`total_sell_qty_at_or_below_P)`. The clearing price is the P that produces the highest matched quantity.
- **Indicative price**: The hypothetical clearing price if the auction were to uncross *right now*, given the current state of submitted orders. It is published periodically during the auction and changes as new orders arrive.
- **Indicative matched volume**: The total quantity that would be filled at the current indicative price.
- **Imbalance**: The net surplus of buy or sell quantity at the current indicative price. If 5,000 buy units and 3,000 sell units would be matched, the imbalance is 2,000 on the buy side. The imbalance signals to sellers that there is excess buy demand not yet being met.
- **Price discovery**: The market mechanism for finding the "correct" price for an asset. Auctions are used at the open and close specifically because they aggregate all available order interest at once, making them more efficient at establishing a reference price than a single order would be.

The indicative price feed allows liquidity providers to respond to imbalances before the uncross. A large buy imbalance may cause the auction to clear higher; a seller can profitably submit sell orders to improve their share of fills and potentially lower the clearing price. Without this feed, participants are blind until the uncross occurs.

Technical details:

- At configurable intervals during the auction phase (`indicative_publish_interval_ms`, default 500ms), compute and publish an `auction.indicative.<symbol>` event.
- Indicative price algorithm: for each candidate price P in the set of all limit prices in the current book, compute `matched_qty(P)`. Select P* = argmax of `matched_qty(P)`. Tie-break rule (multiple P* values): prefer the price closest to the last-traded price; if no last trade exists, prefer the price closest to the midpoint of the highest buy and lowest sell.
- Event payload: `{symbol, indicative_price, indicative_volume, buy_imbalance_qty, sell_imbalance_qty, ts, auction_phas`
- `buy_imbalance_qty = total_buy_qty_at_or_above_P* - matched_qty(P*)`. Equivalently for sell.
- At uncross, the final clearing price must equal the last published indicative price, or the deviation must be explained by a deterministic tie-break rule documented in the spec.

Expected outcome: Participants can observe auction pressure and respond before uncross; the final clearing price is predictable from the indicative feed.

How to verify:

- Hand-calculation test: insert a known set of buy and sell limit orders at specific prices. Compute the expected indicative price manually. Assert the published indicative price matches.
- Imbalance test: construct a book with 500 excess buy units at the indicative price. Assert `buy_imbalance_qty == 500` and `sell_imbalance_qty == 0`.
- Uncross consistency test: assert the final clearing price equals the last published indicative price.
- Cadence test: assert at least one indicative event is published per `indicative_publish_interval_ms` during the auction

phase.

Learning objective: Understand auction mechanics and pre-open/pre-close transparency products.

13.16 Experiment 15: Synthetic Orders Engine (Parent-Child Execution)

Status: Still Valid

Difficulty: Very Complex Estimated effort: 2 to 4 weeks Gap vs real exchange: Institutional workflows rely on synthetic and algorithmic parent-child orders.

Background: The matcher supports several advanced order types but not a full parent orchestration layer for institutional-scale execution.

Financial explanation and motivation:

New terms:

- **Parent order**: A high-level execution intent — for example, "buy 500,000 shares of AAPL over the next 60 minutes". The parent is not itself submitted to the exchange; it is the instruction given to an execution algorithm.
- **Child order**: A small individual order generated by the algorithm from the parent intent and submitted to the exchange. The sum of all child fills should approach the parent quantity over time.
- **Market impact**: The adverse price movement caused by trading. A large order submitted all at once will push prices against the buyer (ask prices rise as the order consumes available sell liquidity). Slicing into children reduces this impact.
- **Signaling risk**: The risk that other participants detect the pattern of systematic child orders and trade ahead of the parent, moving the price before execution completes. A buy child order every 30 seconds at a fixed size is detectable and exploitable.
- **TWAP (Time-Weighted Average Price)**: An execution strategy that divides the parent quantity into equal-sized child orders and submits them at equal time intervals over the execution window. Simple, predictable, and reduces market impact by spreading buying pressure over time.
- **POV (Percentage of Volume, also called Participation Rate)**: An execution strategy that sizes each child order as a fixed percentage of recent observed market volume. If the market is trading 10,000 shares per minute and the POV rate is 10%, each interval targets 1,000 shares. This adapts to market activity rather than using a fixed clock.
- **VWAP (Volume-Weighted Average Price)**: The ratio of total traded value to total traded volume over a period: `VWAP = sum(price_i × qty_i) / sum(qty_i)`. Used as an execution quality benchmark. Executing at or better than VWAP is generally considered a good outcome.
- **Arrival price**: The mid-price (midpoint between best bid and best ask) at the exact moment the parent order was first submitted. Used as a benchmark: did the algorithm cause the price to move adversely from the point of decision?
- **Slippage**: The difference between the arrival price and the average execution price. Positive slippage for a buy means the algorithm paid more than the arrival price, indicating market impact or timing cost.

Technical details:

- Add a `ParentOrderManager` service that subscribes to fill and market data events and maintains parent order state independent of the core engine.
- Parent order fields: `parent_id` (str), `symbol` (str), `side` (str), `total_qty` (int), `strategy` (str: `twap` or `pov`), `start_time` (ms epoch), `end_time` (ms epoch), `pov_rate` (Decimal, for POV only), `max_child_qty` (int, upper cap per child order), `child_limit_offset_ticks` (int, how far from best price to place child limit orders).
- TWAP scheduler: compute `num_intervals = (end_time - start_time) / interval_ms`. Each interval, submit a child limit order for `ceil(remaining_qty / remaining_intervals)` units, priced at `best_opposite_price - child_limit_offset_ticks` (buy) or `+ offset` (sell).
- POV scheduler: each interval, query recent volume from a rolling market data window. Compute `target_qty = round(window_volume * pov_rate)`. Submit child order for `min(target_qty, max_child_qty, remaining_qty)`.
- Child lifecycle: on ACK, record child as active. On FILL, accumulate `filled_qty` on the parent. On child CANCEL (e.g., due to a market halt), re-schedule the quantity.
- Parent terminal states: `COMPLETED` (filled_qty == total_qty), `CANCELLED` (explicit cancel request), `EXPIRED` (end_time reached with unfilled remainder logged as shortfall).
- Emit `parent.progress` each interval: `{parent_id, filled_qty, total_qty, pct_complete, avg_fill_price, arrival_price, slippage_bps}` where `slippage_bps = (avg_fill_price - arrival_price) / arrival_price × 10000` for a buy.

Expected outcome: Large parent intents execute as controlled child streams with measurable execution quality metrics.

How to verify:

- TWAP determinism test: fixed market with no fills, known interval count, assert correct number of child orders submitted at the expected intervals.
- POV scaling test: inject known volume events, assert child sizes scale with market volume at the configured rate.
- Completion test: fill all children; assert parent reaches `COMPLETED` with correct `avg_fill_price`.
- Slippage test: compare `avg_fill_price` to `arrival_price` in a controlled scenario and assert slippage is within expected bounds.
- Stale data test: pause market data for longer than the stale threshold; assert the parent pauses child submission and resumes when data returns.

Learning objective: Understand how high-level execution intent is translated into low-level exchange orders.

13.17 Suggested progression path

1. **Phase 1 (Core)**: Complete Experiments 1, 4, 5, 6, 9, 12, 13, 14, 15 — these cover edge cases and advanced market design with no blockers. (4 and 14 have since shipped — see their own **Status** lines for what, if anything, remains.)
2. **Phase 2 (Hardening)**: Tackle partially-implemented experiments: 2, 3, 7, 8, 10, 11 — these integrate with existing subsystems and add operational maturity.
3. **Phase 3 (New opportunities)**: Experiments 17–20 open new integration and observability tracks.

This progression mirrors how real venues evolve: deterministic matching first, then control frameworks, then advanced market products. It reflects the experiments' original, pre-verification proposal; each experiment's own **Status** line is the authoritative word on what is actually left to build.

Note: Experiment 16 (Self-Trade Prevention) is already implemented in the gateway and matching engine as `SmpAction` modes. It serves as a validation that the roadmap's abstractions align with production needs.

13.18 Experiment 16: Self-Trade Prevention

Status: Already Implemented

Implementation location: `src/edumatcher/models/order.py` (SmpAction enum), `src/edumatcher/engine/order_book.py` (`_sweep()` and `_smp_cancel_resting()`), `src/edumatcher/alf_gwy/gateway.py` and `src/edumatcher/alf_console/main.py` (SMP command syntax).

Why included anyway: Demonstrates that the roadmap's high-level abstractions map cleanly onto production code patterns. Can serve as a learning reference for understanding the matching engine's self-trade enforcement.

Difficulty: Intermediate Estimated effort: — Gap vs real exchange: Every regulated exchange prevents an account from trading against itself.

Background: Currently a gateway can submit both a buy and a sell order at matching prices and they will fill against each other. In production this is prohibited.

Financial explanation and motivation:

New terms:

- **Self-trade**: A trade where the buy order and the sell order belong to the same account. No real economic transfer occurs because the participant is simultaneously on both sides. Inventory and cash net to zero; only the exchange's record of a "trade" changes.
- **Wash trade**: A trade that creates the appearance of market activity but involves no genuine change of ownership or economic risk. Wash trading is illegal in most jurisdictions because it inflates reported volume, misleads other participants about market activity, and can be used to manipulate prices. A self-trade is a form of wash trade.
- **STP (Self-Trade Prevention)**: A mechanism that detects an imminent self-trade at match time and applies a defined action to prevent it. The action is configurable per account or per gateway. Four standard industry modes exist:
  - **Cancel New (CN)**: The incoming order is rejected. The resting order remains in the book unchanged.
  - **Cancel Resting (CR)**: The resting order is cancelled. The incoming order proceeds to match against other participants' orders.
  - **Cancel Both (CB)**: Both the incoming and the resting order are cancelled.
  - **Decrement and Cancel (DC)**: The sizes of the two orders are compared. The smaller is cancelled; the larger

has its quantity reduced by the smaller's quantity. If they are equal, both are cancelled. This minimises disruption while still preventing the self-trade.

STP is mandatory on regulated exchanges. In a simulation it is equally important: without it, a bot that submits orders on both sides inflates its own fill statistics and produces meaningless PnL numbers, defeating the educational purpose of the simulation.

Technical details:

- Add `stp_mode: CN | CR | CB | DC` to gateway or account config. Default: `CN`.
- At match time, before executing any fill, check whether the resting order's `account_id` equals the incoming order's `account_id`. If Experiment 9 (account domain) is not yet implemented, use `gateway_id` as a simpler proxy.
- If a self-trade is detected, apply the configured STP action and do not execute the fill. Emit a `stp.triggered` event with fields: `incoming_order_id`, `resting_order_id`, `account_id`, `stp_mode`, `action_taken` (one of `incoming_cancelled`, `resting_cancelled`, `both_cancelled`, `quantity_adjusted`).
- `stp.triggered` events are published to both the audit channel and the drop-copy channel.
- STP check applies to all order types that can generate fills: Limit, Market, Stop-Limit, Iceberg.

Expected outcome: Self-trades are blocked under all four modes; audit events are produced for every STP action; fills between different accounts are unaffected.

How to verify:

- For each of the four STP modes, set up one resting order and one incoming order from the same account that would self-trade. Assert the correct action is taken per mode.
- DC mode edge case: resting qty 100, incoming qty 150. Assert resting is cancelled and incoming qty is reduced to 50, which then proceeds normally.
- Cross-account test: orders from two different accounts at the same price still fill normally with STP enabled.
- Audit test: assert the `stp.triggered` event contains all required fields.
- No-config test: two accounts with no STP config can trade against each other normally.

Learning objective: Understand wash-trade prevention, regulatory compliance mechanics, and account identity in order routing.

13.19 Experiment 17: Statistics SQL Query CLI Layer

Status: Largely Implemented — verified 2026-09-11

`pm-stats-cli` shipped with thirteen subcommands, among them `daily`, `trades`, `order-events`, `order-lifecycle`, `symbols`, `dates` and `gaps`, and `pm-clearing-cli`, `pm-audit-cli` and `pm-log-cli` cover the clearing, audit and log databases the same way. The background paragraph below was written before any of that existed and is kept for the financial motivation, not as a description of the current state.

`--format csv` has already shipped on `pm-stats-cli`, `pm-clearing-cli` and `pm-audit-cli` (`pm-log-cli` offers only `human`/`json`). Still open from the proposal: a single `daily-pnl` report that joins clearing P&L per gateway — `pnl` and `daily` exist as separate subcommands today, but nothing joins them into one report.

Difficulty: Intermediate Estimated effort: 2 to 3 days Gap vs real exchange: Operators and compliance teams need safe, ergonomic CLI tools to extract analytics without writing raw SQL.

Background: *(As written, before the CLIs existed.)* The statistics database (SQLite) is only accessible via direct SQL queries, which is error-prone and requires database knowledge. Real operations teams use pre-built CLI commands to pull reports by date, gateway, symbol, and time window.

Financial explanation and motivation:

New terms:

- **Report automation**: The process of generating standard operational reports (daily PnL, volume by symbol, gateway activity) on a schedule or on-demand.
- **Audit trail**: Complete record of all orders, fills, and cancellations for regulatory review. Typically filtered by gateway, date range, and symbol.

- **Dashboard data feed**: Time-series metrics (OHLC, volume, imbalance) published to real-time monitoring dashboards.
- **Data governance**: Policies on which operators can query which data, and what time delays apply (e.g., compliance can query historical but not live data).

Without a CLI abstraction, operators either run ad-hoc SQL (risky, slow to learn) or build one-off Python scripts (not scalable). Real exchanges expose ~20–50 named report templates covering daily settlement, regulatory reporting, and performance analytics.

Technical details:

- Add a new `pm-stats-query` or `pm-report` CLI command (or extend `pm-admin-cli` with subcommands).
- Pre-build query templates for common reports:
  - `daily-pnl [--date YYYY-MM-DD] [--gateway GATEWAY]` — gross and net PnL per account.
  - `volume-by-symbol [--date YYYY-MM-DD] [--start-time HH:MM] [--end-time HH:MM]` — total notional and share volume.
  - `gateway-activity [--gateway GATEWAY] [--date YYYY-MM-DD]` — order count, fill rate, rejection rate.
  - `fills-by-symbol [--symbol SYMBOL] [--date YYYY-MM-DD]` — OHLC, volume, imbalance.
  - `audit-trail [--gateway GATEWAY] [--date YYYY-MM-DD] [--symbol SYMBOL]` — full order lifecycle log.
- Output format options: `--format json | csv | table` (table is the default for human readability).
- Queries execute against the SQLite stats database; no custom aggregation logic needed — the DB schema already supports these views.
- Add optional `--limit N` to cap result row count (default: 10,000).

Expected outcome: Operators can run `pm-report daily-pnl --date 2026-06-14 --format csv` and get a CSV without writing SQL. Audit queries become repeatable and auditable.

How to verify:

- Integration test: run each template query and assert output schema matches expected columns.
- End-to-end test: load sample session data, run a daily-pnl query, verify sum of gateway PnL matches engine's reported total.
- Output format test: run with `--format json` and `--format csv` and assert parseable output.
- Regression test: verify raw SQL equivalent queries return identical results.

Learning objective: Understand operational reporting patterns, data access abstraction, and human-centered CLI design.

13.20 Experiment 18: Drop Copy Durability and Restart Recovery

Status: Partially Implemented — verified 2026-09-11

The *live* half exists: `drop_copy.replay.{recipient_id}` lets a subscriber recover a gap without a file, and `pm-dc-spy` is the tail-and-inspect tool the proposal calls `dropcopy_tail`. What is genuinely still open is the **durability** half this experiment is named for — there is no `data/dropcopy_YYYYMMDD.log`, no immutable rename at session close, and no sequence recovery from a file at startup. That is the interesting part of the exercise, and it is untouched.

Difficulty: Intermediate Estimated effort: 2 to 4 days Gap vs real exchange: Production drop-copy systems survive process restarts and network outages.

Background: The current drop-copy buffer is in-memory and bounded (10,000 messages). If the engine restarts, all buffered events are lost. Participants need to re-bootstrap their view of positions from trade history, which is slow and error-prone during high-volume sessions.

Financial explanation and motivation:

New terms:

- **Durable log**: An append-only file written to disk, guaranteeing that events persist across process restarts.
- **Session rotation**: A new log file per trading session (e.g., `dropcopy_20260614.log`), simplifying rotation and archival.
- **Replay window**: The newest N messages available for replay without querying the archive. Typical: last 1 hour of events.
- **Gap tolerance**: How long a subscriber can be offline before gap detection kicks in. Typical: 15–60 minutes.
- **Restart sequence**: On engine restart, drop-copy publisher re-initializes and continues publishing with sequence numbers reset to 1.

Technical details:

- Write all drop-copy events to a durable JSON-lines log file: `data/dropcopy_YYYYMMDD.log`.
- One event per line: `{"seq": 1, "timestamp": 1718000000000000000, "gateway_id": "TRADER01", "event_type": "order.fill", ...}`.
- At session end (when engine receives CLOSED state transition), rename file to `dropcopy_YYYYMMDD.HHMMSS.CLOSED.log` to mark it immutable.
- On engine startup, scan for the most recent unclosed log file. If found, initialize drop-copy sequence counter to `max_seq + 1` from that file.
- Add `dropcopy_tail` utility: read the last N lines of the log file and re-publish them on the drop-copy socket for slow subscribers to catch up.
- Add `dropcopy_reconcile` utility: read two log files (local participant log vs. engine archive), align by sequence number, and report any gaps or duplicates.

Expected outcome: Drop-copy events survive engine restarts. Participants can query historical drop-copy by session date and get exact sequence continuity.

How to verify:

- Persistence test: write 100 events, restart engine, verify next 10 events have sequence numbers 101–110 (no reset).
- Rotation test: run through a full session, verify CLOSED log file is created and new session starts with fresh sequence.
- Recovery test: simulate a participant re-connecting 1 hour after disconnect; run `dropcopy_tail` and verify they receive the last 50 events.
- Gap detection test: delete one line from a log file, run `dropcopy_reconcile`, assert it reports exactly one gap at the correct sequence.

Learning objective: Understand durability patterns, log rotation, and participant recovery workflows in high-availability systems.

13.21 Experiment 19: Cross-Host Operational Validation Matrix

Status: Still Valid — verified 2026-09-11

Cross-host deployment itself is designed and documented (`docs-design/EduMatcher-Cross-host-connection.md`, and the bind/connect split in `config.py`), but the *validation matrix* this experiment asks for — the same scenario replayed under single-host, LAN and latency-injected WAN topologies, with the final book state compared across all three — does not exist. That comparison is the whole point of the experiment.

Difficulty: Intermediate Estimated effort: 2 to 3 days Gap vs real exchange: Multi-host deployments must pass deterministic end-to-end tests before going live.

Background: The runtime now supports configurable data and config paths plus cross-host endpoint resolution. A multi-host exchange (engine on host A, gateways on hosts B and C) must maintain order determinism and message ordering across network boundaries. The validation matrix ensures this works.

Financial explanation and motivation:

New terms:

- **Latency parity**: Behavior remains deterministic even when network latencies between hosts vary.
- **Message ordering guarantee**: Orders submitted by different gateways are processed in submit-timestamp order, not arrival order, so latency does not affect fills.
- **Startup sequence**: The order in which processes bind sockets and wait for others to connect. If violated, processes hang or fail.
- **Network fault injection**: Simulate packet loss, reordering, or delays to validate that the exchange does not diverge from its single-host behavior.

Technical details:

- Create a test matrix that runs the same 100-order scenario under three topologies:
  1. **Single-host** (baseline): all processes on localhost, default ZMQ configuration.
  2. **LAN multi-host**: engine on 10.0.0.10, gateway on 10.0.0.11, ~1ms latency (network overhead).
  3. **WAN multi-host** (simulated): engine on 10.0.0.10, gateway on 10.0.0.11 + 50ms injected delay via tc (traffic

control).
- For each topology, verify:
  - Final order book state is identical across all topologies.
  - Fill prices and quantities match exactly.
  - Drop-copy sequence numbers are contiguous (no gaps or duplicates).
  - Audit log is byte-identical (same topic order, same event payloads).
- Add a `stress_cross_host.sh` script that runs all three topologies back-to-back with different random seeds and reports pass/fail.

Expected outcome: Operators gain confidence that cross-host deployments will not diverge from single-host behavior due to network variations.

How to verify:

- Topology equivalence test: run the same scenario on single-host and LAN multi-host, hash the audit logs, assert hashes match.
- Latency robustness test: inject 50–200ms delays via network emulation and re-run; audit logs should still match.
- Fault injection test: drop 0.1% of messages at the network layer (using tc or netcat), verify engine detects and reports the gap or reconnect.
- Determinism test: run 10 times with the same seed on each topology; assert all 10 runs produce identical results on each topology (but not necessarily identical between topologies if latencies differ).

Learning objective: Understand distributed systems testing, latency-robust message ordering, and operational validation patterns.

13.22 Experiment 20: Installed-Mode vs. Source-Mode Parity Validation

Status: Still Valid — verified 2026-09-11

`DATA_DIR` resolution distinguishes the two modes today (`config.py::_resolve_data_dir`, described in Dev practice), and `pm-setup` bootstraps the installed layout — but there is no `test_mode_parity.py` asserting that the two modes produce identical audit logs and book state. The mechanism exists; the proof does not.

Difficulty: Basic Estimated effort: 1 to 2 days Gap vs real exchange: Installation and setup are common operational concerns; deployment paths must be equivalent.

Background: EduMatcher supports two run modes: **source mode** (Poetry + git clone) and **installed mode** (pipx + `pm-setup`). Both should behave identically. Currently, there is no automated test that validates parity across these modes.

Financial explanation and motivation:

New terms:

- **Source mode**: Developer runs `poetry run pm-engine` from a git clone.
- **Installed mode**: End user runs `pm-setup`, then `pm-engine` from PATH after `pip install` (or pipx).
- **Configuration parity**: Same config files, same environment variable resolution, same default paths.
- **Reproducibility**: Users should get the same behavior regardless of which installation method they choose.

Technical details:

- Create a `test_mode_parity.py` test suite that:
  1. Starts the engine in **source mode** (Poetry), runs a 50-order scenario, saves the audit log and final book state.
  2. Installs the package with `pipx install .` (or equivalent).
  3. Runs `pm-setup --data-dir /tmp/test-installed` to initialize the installed-mode directory.
  4. Starts the engine in **installed mode** using the same config, runs the same 50-order scenario.
  5. Compares audit logs and book state between modes; they should match exactly.
- Test both configuration resolution paths:
  - Via the deployed copy under `EDUMATCHER_DATA_DIR`.
  - Via default fallback (current directory, then `~/.local/share/edumatcher`).
- Verify data directory resolution:
  - Via `EDUMATCHER_DATA_DIR` env var.
  - Via default fallback.

Expected outcome: CI pipeline gains a gate that prevents installed mode regressions. Users are confident that `pipx install` works correctly.

How to verify:

- Build parity test: install edumatcher with `pipx`, run test suite, assert it passes.
- Config path test: test that `EDUMATCHER_DATA_DIR` relocates the deployed config along with every other data file, in both modes.
- Audit log equivalence: run test scenario in both modes, assert audit logs are byte-identical.
- Performance parity: measure latency per order in both modes; should differ by $< 5\%$.

Learning objective: Understand packaging, installation, and deployment verification best practices.

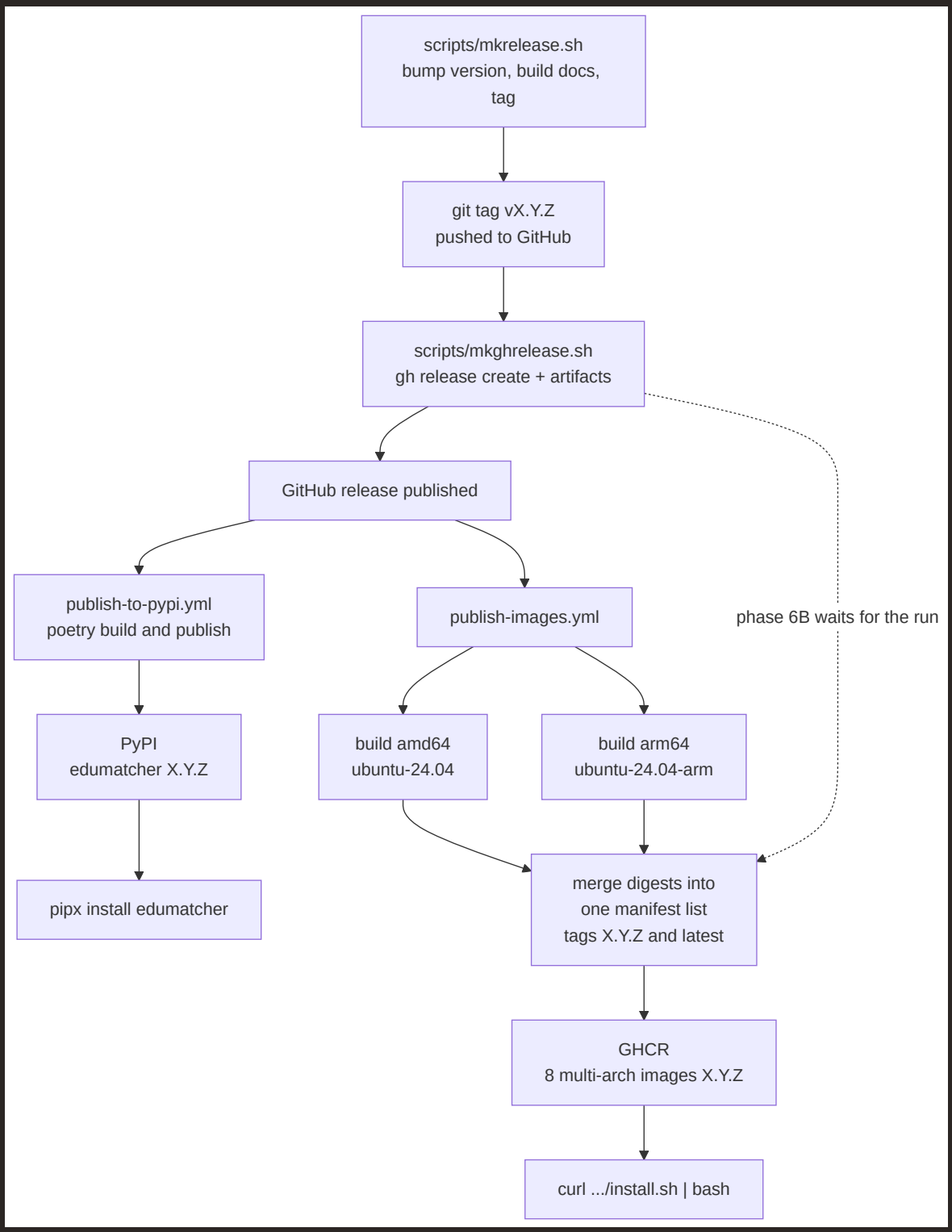
CHAPTER 14

The Release Process

How a version of EduMatcher is cut and published: the scripts and workflows involved, the developer checklist, and the pitfalls that catch newcomers. The day-to-day development loop that precedes a release is in Development Practice.

14.1 How a release is produced

Two scripts and two GitHub workflows. One tag produces the Python package and the container images together, all carrying the same version, which is what lets the installer pin a whole system with a single number.



Each image is built **natively** on both architectures rather than emulated, then the two are joined into one manifest list. A user on Intel and a user on Apple Silicon pull the same tag and each gets the right binary.

The eight published images are:

```
ghcr.io/johan162/edumatcher                the exchange, all pm-* processes
ghcr.io/johan162/edumatcher-ai-sim         the AI-trader simulation runner
ghcr.io/johan162/edumatcher-terminal-gui   TapeDeck (pm-terminal)
ghcr.io/johan162/edumatcher-log-gui        the Log Operator Console
ghcr.io/johan162/edumatcher-config-gui     the Configuration GUI
ghcr.io/johan162/edumatcher-trader-gui     the Trading GUI
ghcr.io/johan162/edumatcher-book-gui       the order-book viewer
ghcr.io/johan162/edumatcher-dash-gui       the Operator Dashboard
```

`latest` is only moved for an exact `vMAJOR.MINOR.PATCH` tag, so a pre-release never becomes what a new user gets by default — the same rule the PyPI workflow uses to choose between PyPI and TestPyPI.

14.1.1 Publishing by hand

`make ghcr-push` in `deployment/docker/` builds the seven images other than `edumatcher-ai-sim` from your checkout and pushes them, for when the workflow cannot run:

```
export GITHUB_USER=<you> GHCR_TOKEN=<token with write:packages>
make ghcr-push                              # all six, tagged :dev
make ghcr-push TAG=0.20.6 FORCE=1 LATEST=1  # as a release tag
```

It builds only for the architecture you are on. Pushing a single-architecture image over a release tag replaces the manifest list, and users on the other architecture then get "no matching manifest" — which you will not notice, because your own machine keeps working. That is why a release-looking tag needs `FORCE=1` and why `latest` is never moved unless asked.

14.2 Developer release checklist

For the maintainer cutting a release. Steps 1-2 are local, 3-5 are automated but need watching, and 6-7 are the checks that the release actually works for somebody who is not you.

14.2.1 Before tagging

1. **Working tree is clean and tests pass.**

   ```
   ./scripts/mkbld.sh
   ```

2. **`CHANGELOG.md` has an entry for this version.**

- Add a new `CHANGELOGENTRY.md`. Using the custom copilot skill `/changelog-entry` to create a draft version based on the git-logs
- or if there is no copilot available Use the script `scripts/mkchlogentry.sh` drafts one from the commit logs.

14.2.2 Tag and release

3. **Run `scripts/mkrelease.sh`.** It bumps the version, builds the documentation bundles, commits and tags.
4. **Wait for the CI workflows on the tag to go green** before creating the release.
5. **Run `scripts/mkghrelease.sh`.** It validates the artifacts in `dist/`, creates the GitHub release, and then waits for the container image workflow.

Option	Effect
`--dry-run`	Show what would happen; create nothing
`--pre-release`	Force pre-release marking regardless of the tag
`--skip-images`	Do not wait for the image workflow
`IMAGE_WAIT_MINUTES=n`	How long to wait (default 30)

```
git switch main && git pull --ff-only
./scripts/mkghrelease.sh
```

If the image workflow fails, the GitHub release still exists — only the images are missing. Re-run just that part:

```
gh run view <run-id> --log-failed
gh workflow run publish-images.yml -f tag=vX.Y.Z
```

14.2.3 After the release

6. **Verify the one-line install as a stranger would.** First **stop any stack you already have running** — the released deployment and the source-built one use the same container names and host ports, so an install started beside a running stack silently attaches to it and verifies nothing:

```
make -C deployment/docker down-all
```

Then install into a throwaway directory so your own instance is untouched:

```
curl -fsSL https://raw.githubusercontent.com/johan162/EduMatcher/vX.Y.Z/deployment/curl/install.sh \
    | bash -s -- --dir /tmp/em-release-test
```

Then open http://localhost:8090, and clean up with `cd /tmp/em-release-test && ./edumatcher.sh uninstall --data`.

7. **Verify the PyPI install** in a fresh environment: `bash  pipx install edumatcher==X.Y.Z`

Where releases usually go wrong

Two failures are quiet rather than loud. An image built from PyPI instead of the checkout looks like a successful build but ships the *previous* release — step 5's image-workflow log should show `Installing local wheel` (the line the Dockerfile prints when it finds a wheel staged by the build, rather than installing from PyPI). And private GHCR packages fail only for other people, never for the maintainer who is already authenticated — step 6, done as a stranger with no credentials, is what catches that.

14.3 Summary: To make a release, step-by-step

```
# 1. Bump version in pyproject.toml
poetry version 0.3.2

# 2. Create the changelog template with help of copilot using the custom skill
/changelog-entry 0.3.2 patch

# 3. Review and edit CHANGELOG.md as needed. The commit the new changelog
git add CHANGELOG.md
git commit -m "chore(changelog): v0.3.2"

# 4. Build and validate release artifacts.
# Use --intro for real releases because mkghrelease.sh expects the intro bundle.
./scripts/mkbld.sh --intro

# 5. Commit release-ready changes on develop, then preview release actions
GITHUB_USER=<your-gh-user> ./scripts/mkrelease.sh patch --dry-run

# 6. Execute the local release flow (squash merge develop -> main, tag, push, sync back)
GITHUB_USER=<your-gh-user> ./scripts/mkrelease.sh patch

# 7. After CI is green on main, create the GitHub release from main
git switch main && git pull --ff-only
./scripts/mkghrelease.sh
```

14.4 Appendix: The release scripts

14.5 What `mkbld.sh` does

`mkbld.sh` is the build gate before release scripts. It currently:

1. validates environment and required Poetry tools
2. runs `black`, `flake8`, `mypy` (and `pyright` if available)
3. runs `npm install`, `npm run typecheck`, `npm run lint`, and `npm test` for each app in `web-apps/`, with up to three apps in parallel
4. runs pytyest tests with coverage threshold **80%**
5. updates coverage badge when `coverage.xml` changed
6. builds and verifies Python packages
7. builds all documentations, user-guide PDF+EPUB, doc-site, training bundles, etc
8. optionally builds Exchange Intro bundle when `--intro` is provided

For release publishing, prefer running `./scripts/mkbld.sh --intro`.

14.5.1 What `mkrelease.sh` expects

The script is designed around this model:

- `GITHUB_USER` environment variable is set
- you are on a **clean `develop` branch**
- local `develop` is synced with remote
- the requested version is not already tagged
- `CHANGELOG.md` already contains an entry for that version
- release artifacts already exist in `dist/` and pass `twine check`

Important usage detail: `mkrelease.sh` argument is only the release type (`major`, `minor`, or `patch`). The version is read from `pyproject.toml`.

During execution, `mkrelease.sh` will:

1. validate repo state and changelog/version/tag preconditions
2. squash-merge `develop` into `main` and create `v<version>` tag
3. push `main` and the tag
4. merge `main` back into `develop` and push `develop`
5. wait for GitHub Actions completion with `gh run watch --exit-status`

14.5.2 What `mkghrelease.sh` expects

The script is designed around this model:

- `GITHUB_USER` environment variable is set
- authenticated `gh` CLI on PATH
- you are on a **clean `main` branch** synced with remote
- latest release tag already exists on `main`
- required artifacts exist:
  - wheel in `dist/`
  - sdist in `dist/`
  - user-guide bundle in `docs/dist/`
  - exchange-intro bundle in `docs-exchange-intro/dist/`

It auto-detects pre-releases from tags ending in `rcN` (or you can force with `--pre-release`) and creates the GitHub release using notes extracted from `CHANGELOG.md`.

14.6 Common pitfalls for new developers

14.6.1 Forgetting which data directory is live

In a source checkout, `src/data/` — not `~/.local/share/edumatcher/` — is the data directory every local process and every test reads and writes. A "compiled config has been modified since it was compiled" error from `pytest`, or a `pm-config-deploy` you ran that doesn't seem to take effect, is almost always this. See Where your data directory actually is above.

14.6.2 Starting clients before the engine

Most processes assume the engine sockets already exist. Start `pm-engine` first.

14.6.3 Using a gateway ID that is not in `engine_config.yaml`

`pm-alf-console --id SOMEONE` only works if that gateway is configured.

14.6.4 Forgetting that observer processes depend on each other

`pm-ticker` and `pm-board` rely on statistics written by `pm-stats`.

14.6.5 Assuming docs-only changes need no validation

They still need:

```
poetry run mkdocs build
```

14.6.6 Treating helper scripts as canonical truth

Some scripts are polished automation; others still show drift from earlier repo history. Read before relying on them.

14.6.7 Ignoring end-to-end behavior

A change that passes unit tests can still break:

- startup order
- message topics
- gateway acknowledgements
- persistence restore
- UI observers

That is why a minimal live run is worth doing.

14.7 Suggested first-week path for a new developer

If you are onboarding, this is a good sequence:

1. Set up Poetry and build the docs
2. Run the minimal live system once
3. Submit a few manual orders through `pm-alf-console`
4. Run the normal test suite
5. Read the deterministic verification page
6. Pick one small bug fix or documentation improvement
7. Run the full quality gate before opening a PR

That path teaches both the **theory** and the **operational shape** of the system before you attempt deeper engine work.