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

PARTICIPANT GUIDE

A Guide to becoming a Trader using The EduMatcher Trading System

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 Participant 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.

Participant Guide

`edumatcher`, Version: v0.77.0

Trading, market making, and participant-facing tools

Johan Persson <johan162@gmail.com>

Generated on November 8, 2026

Chapters

1 How to Use This Book 1

2 The Order Book 3

3 A Full Trading Day 9

4 Gateways and How You Connect 17

5 The Trader Console (`pm-alf-console`) 23

6 Order Types 47

7 Combo Orders 81

8 Implied Orders (a Concept Not Yet Implemented) 93

9 Market-Maker Quotes 101

10 Market Making 107

11 The Market-Maker Bot (`pm-mm-bot`) 143

12 AI Traders 175

13 How the AI Traders Decide 181

14 Positions and P&L 187

15 Your End-of-Day Statement 221

16 Trader Information Terminal — "TapeDeck" (`pm-terminal`) **227**

17 Trading GUI (`pm-trading-ui`) **245**

18 Order Book Viewer (`pm-book`) **289**

19 FAQ: Orders, Sessions and Positions **309**

Contents

1 How to Use This Book **1**
- 1.1 How the book is organized . . . 1
- 1.2 Where to start . . . 2

2 The Order Book **3**
- 2.1 What is an order book? . . . 3
- 2.2 A concrete order book . . . 5
- 2.3 The bid-ask spread . . . 5
- 2.4 Book depth . . . 5
- 2.5 Price-time priority . . . 6
- 2.6 Step-by-step: what happens when orders match . . . 7
- 2.7 Passive vs. aggressive orders . . . 7
- 2.8 Key terms summary . . . 7
- 2.9 Implementation Note: Tick Prices . . . 8

3 A Full Trading Day **9**
- 3.1 The shape of a trading day . . . 9
- 3.2 The five phases . . . 10
- 3.3 Enabling or disabling session handling . . . 10
- 3.4 Specifying time for the daily session . . . 10
- 3.5 Phase 1 — PRE_OPEN . . . 11
- 3.6 Phase 2 — OPENING_AUCTION . . . 11
- 3.7 Phase 3 — CONTINUOUS . . . 13
- 3.8 Phase 4 — CLOSING_AUCTION . . . 13
- 3.9 Phase 5 — CLOSED . . . 13
- 3.10 What happens to my orders at each transition? . . . 13
- 3.11 Triggering phase transitions in EduMatcher . . . 14
- 3.12 Putting it all together — a full-day scenario . . . 14

4 Gateways and How You Connect **17**
- 4.1 What is a gateway? . . . 17
- 4.2 Industry-standard gateway protocols . . . 18
- 4.3 EduMatcher's gateways . . . 19
- 4.4 One user per gateway — a learning simplification . . . 20

4.5 Multiple gateways and arrival order . . . 21
4.6 See also . . . 22

5 The Trader Console (`pm-alf-console`) 23
5.1 `pm-alf-console` is not a gateway . . . 23
5.2 What this process does . . . 24
5.3 Architecture position . . . 24
5.4 When to use ALF — protocol comparison . . . 26
5.5 Starting `pm-alf-console` . . . 26
5.6 Session lifecycle . . . 28
5.7 Command Format . . . 29
5.8 Reconnect and state re-sync playbook . . . 41
5.9 Gateway Responses . . . 42
5.10 End-to-end operator walkthrough . . . 44
5.11 Multiple `pm-alf-console` Sessions . . . 44
5.12 Interactive Features . . . 45
5.13 Common mistakes and fast triage . . . 45
5.14 See also . . . 46

6 Order Types 47
6.1 Contents . . . 47
6.2 MARKET . . . 50
6.3 LIMIT . . . 52
6.4 STOP (Stop-Market) . . . 53
6.5 STOP_LIMIT . . . 54
6.6 FOK (Fill-or-Kill) . . . 55
6.7 ICEBERG . . . 57
6.8 COMBO . . . 58
6.9 Time-in-Force (TIF) . . . 61
6.10 Stop Trigger Logic . . . 62
6.11 IOC (Immediate-Or-Cancel) . . . 65
6.12 TRAILING_STOP . . . 67
6.13 Order Amendment (AMEND) . . . 71
6.14 OCO (One-Cancels-Other) . . . 74
6.15 See also . . . 79

7 Combo Orders 81
7.1 What Is a Combo Order, and Why Would I Use One? . . . 81
7.2 What Are Combo Orders? . . . 82
7.3 Why Do Combo Orders Exist? . . . 82
7.4 New Strategies Enabled by Combos . . . 83
7.5 Complexity Introduced by Combos . . . 84
7.6 Trading Combos in EduMatcher . . . 86
7.7 New Demands on Market Makers . . . 87
7.8 Example Strategies in Detail . . . 89
7.9 Summary . . . 90
7.10 See also . . . 90

8 Implied Orders (a Concept Not Yet Implemented) 93

8.1 The key idea in one paragraph . . . 93
8.2 First: beginner terminology . . . 94
8.3 Why combo markets are hard without implied orders . . . 94
8.4 What problem implied orders solve . . . 95
8.5 How implied orders accomplish this . . . 95
8.6 Detailed calculation model (2-leg spread) . . . 95
8.7 Generalized weighted synthetic pricing . . . 96
8.8 Worked Example 1 (2-leg, equal ratios) . . . 97
8.9 Worked Example 2 (3-leg with ratios) . . . 98
8.10 Directions of implication . . . 99
8.11 Practical implementation challenges . . . 100
8.12 How this relates to EduMatcher today . . . 100

9 Market-Maker Quotes **101**
9.1 Concept . . . 101
9.2 Startup Quote Seeding . . . 101
9.3 Fills Against Seed Quotes Before the MM Connects . . . 103
9.4 Gateway Commands . . . 104
9.5 Message Flow . . . 105
9.6 Refresh Policies . . . 105
9.7 Disconnect And Risk Controls . . . 106

10 Market Making **107**
10.1 What is a market maker? . . . 107
10.2 Configuring a market-maker gateway . . . 107
10.3 The QUOTE command . . . 108
10.4 Quote lifecycle . . . 109
10.5 Quote refresh policy . . . 110
10.6 MM obligations enforcement . . . 113
10.7 Market-Maker Protection (MMP) — data model only, not enforced by the engine . . . 114
10.8 Cancelling a quote . . . 114
10.9 Startup seeding — pre-loading quotes from config . . . 114
10.10 Disconnect behaviour . . . 120
10.11 Full worked example . . . 121
10.12 Config reference summary . . . 121
10.13 MM quote identification and quote-leg mapping . . . 122
10.14 See also . . . 140

11 The Market-Maker Bot (`pm-mm-bot`) **143**
11.1 What is pm-mm-bot? . . . 143
11.2 Quick start . . . 145
11.3 Gateway identity convention . . . 145
11.4 Architecture . . . 146
11.5 Bot lifecycle . . . 146
11.6 Pricing logic . . . 148
11.7 Quote refresh logic . . . 149
11.8 Session state handling . . . 150
11.9 Bootstrap: starting with an empty book . . . 150

11.10 Per-symbol settings . . . 150
11.11 CLI reference . . . 152
11.12 Config file . . . 154
11.13 Engine configuration . . . 167
11.14 Usage examples . . . 169
11.15 Understanding bot output . . . 170
11.16 Troubleshooting . . . 171
11.17 See also . . . 173

12 AI Traders **175**
12.1 What are the AI trader processes? . . . 175
12.2 Presets . . . 176
12.3 Running one AI trader . . . 177
12.4 Running a swarm . . . 178
12.5 Your own preset . . . 179
12.6 Reading its output . . . 179
12.7 See also . . . 179

13 How the AI Traders Decide **181**
13.1 The loop . . . 181
13.2 Strategies: what to trade . . . 182
13.3 Execution styles: how to trade it . . . 182
13.4 Tempo: how often and how big . . . 183
13.5 Risk: what it will not do . . . 184
13.6 Writing a preset . . . 184
13.7 The other bot: `pm-mm-bot` . . . 185

14 Positions and P&L **187**
14.1 What this page covers . . . 187
14.2 Starting the clearing process . . . 188
14.3 Data folder location . . . 192
14.4 SQLite database schema . . . 192
14.5 Position tracking . . . 197
14.6 VWAP average cost . . . 198
14.7 Realized P&L . . . 199
14.8 Unrealized P&L . . . 199
14.9 Worked example . . . 199
14.10 Position arithmetic in depth . . . 200
14.11 Using pm-clearing-cli . . . 204
14.12 Clearing cookbook — common questions and answers . . . 217
14.13 Notes . . . 218
14.14 Quick-reference: P&L formulas . . . 219
14.15 See also . . . 219

15 Your End-of-Day Statement **221**
15.1 What the statement is . . . 221
15.2 An example, line by line . . . 221
15.3 Where today's number comes from . . . 223
15.4 Questions . . . 224

15.5 Words used on this page . . . 224

16 Trader Information Terminal — "TapeDeck" (`pm-terminal`) 227
16.1 Overview . . . 227
16.2 Prerequisites . . . 228
16.3 Running the application . . . 228
16.4 A tour of the interface . . . 234
16.5 Screen tour . . . 236
16.6 Configuration reference . . . 241
16.7 Troubleshooting . . . 243
16.8 Related documentation . . . 244

17 Trading GUI (`pm-trading-ui`) 245
17.1 What the Trading GUI is . . . 245
17.2 How it talks to the exchange . . . 246
17.3 Prerequisites . . . 247
17.4 Running the application . . . 248
17.5 Roles and access rights . . . 251
17.6 The application shell . . . 252
17.7 Screens available to every role . . . 254
17.8 The TRADER role . . . 258
17.9 The MARKET_MAKER role . . . 268
17.10 The ADMIN role . . . 271
17.11 Errors and how to read them . . . 275
17.12 Typical workflows . . . 280
17.13 Keyboard shortcuts and power-user mode . . . 282
17.14 Settings . . . 283
17.15 Configuration reference . . . 284
17.16 Troubleshooting . . . 286
17.17 Related documentation . . . 288

18 Order Book Viewer (`pm-book`) 289
18.1 Overview . . . 289
18.2 Prerequisites . . . 290
18.3 Running the application . . . 291
18.4 The Compose setup . . . 297
18.5 A tour of the interface . . . 298
18.6 Connection states . . . 302
18.7 Configuration reference . . . 303
18.8 Checking the bridge . . . 305
18.9 Troubleshooting . . . 305
18.10 Related documentation . . . 306

19 FAQ: Orders, Sessions and Positions 309
19.1 Order types . . . 309
19.2 Positions and P&L . . . 310
19.3 Saved orders . . . 310
19.4 Self-match prevention . . . 311
19.5 Prices . . . 311

CHAPTER 1

How to Use This Book

The Participant Guide is about **taking part in the market**: understanding the order book and the trading day, placing and managing every kind of order, quoting as a market maker, running trading bots, reading your positions and profit, and using the trading screens. It is written from the trader's seat.

Who it is for	Students and traders, and instructors playing a participant in a class
What it assumes	You have finished the Quick Start Guide: you can start the exchange and connect a trader console
What it leaves to other books	Configuring symbols, risk limits and schedules (Operator's Guide); wire formats (Protocols and Clients); every command-line option (Reference Manual)

1.1 How the book is organized

Part	What you learn
I. Trading basics	How an order book works, what each phase of the trading day does to your orders, and how you connect
II. Orders	The trader console, every order type and time in force, combo orders, and the idea of implied orders
III. Market making	What a market maker does, quote obligations and lifecycle, and the market-maker bot
IV. Automated trading	The AI trader bots, and how they decide what to trade
V. Positions and results	Position tracking, average cost, realized and unrealized P&L

Part	What you learn
VI. Front-ends	TapeDeck, the Trading GUI and the Order Book Viewer
Appendix	An FAQ on order types, saved orders, average cost and self-match prevention

1.2 Where to start

- **New to trading:** read Part I in order — especially The Order Book. Everything else builds on it.
- **Comfortable with markets:** start at The Trader Console and Order Types, and dip into the rest as questions come up.
- **Prefer exercises:** the Training Guide covers the same ground as hands-on lessons and links back here for the explanations.

Who configures what

Many rules you meet as a trader — price collars, order-size limits, circuit breakers, the trading timetable, market-maker obligations — are set by the operator in the exchange's configuration. This book explains how they affect you; the Operator's Guide explains how they are set.

CHAPTER 2

The Order Book

Learning objectives

After reading this page you will understand:

- What an order book is and how it is structured
- The difference between a bid and an ask
- What the spread is and why it matters
- How price-time priority determines who gets filled first
- What happens step-by-step when two orders match

2.1 What is an order book?

When you buy a share of stock, someone else must sell it to you. But they may not be available at exactly the moment you want to trade, and you may not agree on price. The **order book** is the mechanism that bridges this gap.

An order book is the live, sorted list of currently executable resting buy and sell orders for one instrument. Conditional orders such as stops are tracked separately until they trigger into executable orders. Every tradable instrument has its own book. Orders are separated into two sides:

Side	Also called	What it represents
Bids	Buy side	Traders who want to buy, sorted highest price first
Asks	Offer / sell side	Traders who want to sell, sorted lowest price first

The matching engine continuously checks whether the best bid and the best ask overlap in price. When they do, a **trade** is produced.


ORDER
SYMBOL: AAPL (Apple Inc.)
BUY ORDERS (BIDS)
PRICE (BID)
QUANTITY
CUM. Q
$174.98
10
$174.97
40
$174.96
200
$174.95
200
$174.93
300
$174.92
50
$174.91
50
$174.90
50
BEST BID:
$174.98
(HIGHEST BUY PRICE)
BUY SIDE
AGGREGATED BIDS:
Total size available to se
into at different prices.

Figure 1: The orderbook.

2.2 A concrete order book

Imagine AAPL is trading around $150. The live book might look like this:

```
BID (buy orders)          ASK (sell orders)
──────────────────        ──────────────────

Qty    Price              Price    Qty
───    ─────              ─────    ───
200    149.90             150.10    100
500    149.80             150.20    300
150    149.70             150.30    500
300    149.60             150.50    200
```

- The **best bid** (highest buy price) is **149.90** for 200 shares.
- The **best ask** (lowest sell price) is **150.10** for 100 shares.
- These prices do not overlap — no trade happens yet.

2.3 The bid-ask spread

The **spread** is the gap between the best bid and best ask:

$$\text{spread} = \text{best ask} - \text{best bid} = 150.10 - 149.90 = 0.20$$

The spread represents the cost of an immediate round-trip (buy then sell). A trader who buys at the ask ($150.10) and immediately sells at the bid ($149.90) loses $0.20 per share.

Narrow spread (e.g. $0.01) — a **liquid** market. Many participants are competing to trade, so prices are tight. Cheap to trade in and out.

Wide spread (e.g. $2.00) — an **illiquid** market. Few participants, or high uncertainty. Expensive to trade in and out.

The **mid-price** is the midpoint between bid and ask:

$$\text{mid} = \frac{149.90 + 150.10}{2} = 150.00$$

2.4 Book depth

Depth refers to how much volume (quantity of shares) is available at or near the top of each side. A deep book absorbs large orders without moving the price much. A thin book has little resting quantity and even a moderate order can push the price significantly.

Looking at our example book above, if a trader submits a market sell order for 600 shares, it would: 1. Fill 200 shares at the best bid of **149.90** 2. Fill 400 shares at the next level, **149.80**

The average fill price would be:

$$\frac{(200 \times 149.90) + (400 \times 149.80)}{600} = \frac{29980 + 59920}{600} = \frac{89900}{600} \approx 149.83$$

The trader wanted to sell at ~149.90 but the large size caused the actual average to be 149.83. This difference is called **slippage**.

2.5 Price-time priority

When multiple orders rest at the same price, the matching engine must decide which one fills first. EduMatcher uses **price-time priority** (also called FIFO priority):

1. **Price first** — the order with the best price for the counterparty fills first. For bids: the highest price wins. For asks: the lowest price wins.
2. **Time second** — among orders at the same price, the one that arrived earliest fills first.

2.5.1 Worked example — three bids at the same price

Suppose three traders each submitted a LIMIT BUY at exactly $149.90:

Order	Trader	Time submitted	Qty
A	GW01	09:30:01.100	100
B	GW02	09:30:01.250	200
C	GW03	09:30:03.900	150

Now a LIMIT SELL at $149.90 for 250 shares arrives. The engine processes bids in time order (A before B before C):

1. **Order A** (GW01, 100 shares) fills first — submitted earliest.
2. **Order B** (GW02, 200 shares) — only 150 more needed, so 150 of the 200 fills, leaving 50 shares of Order B resting.
3. **Order C** (GW03) — gets nothing this time; it was last in queue.

The result:

Order	Fill	Remaining
A	100	0 (fully filled)
B	150	50 (partially filled, still resting)
C	0	150 (still resting, behind Order B in queue at this price)

This is why being early matters: all else being equal, the first order in queue has an advantage. This creates an incentive for market participants to submit orders quickly — one driver of the speed race in modern markets.

2.6 Step-by-step: what happens when orders match

Here is the full lifecycle of a trade between a resting limit order and an incoming aggressive order.

State: AAPL book has a resting BID from GW01

```
Resting bid: GW01 BUY 100 @ 150.00 (LIMIT, arrived at 09:31:00)
```

Action: GW02 submits a SELL LIMIT at 150.00 for 100 shares

```
NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=150.00
```

Engine processing:

1. Incoming order hits the matching engine via ZeroMQ PUSH socket.
2. Engine checks: is there a resting bid at or above $150.00? Yes — GW01 at $150.00.
3. Trade price is the **resting order's price** ($150.00) because it was first.
4. Engine generates a `trade.executed` event:
   - buyer: GW01, qty: 100, price: 150.00
   - seller: GW02, qty: 100, price: 150.00
5. Both orders are fully filled and removed from visible bid/ask levels.
6. Engine publishes `order.fill.GW01` and `order.fill.GW02` to both gateways.
7. Engine publishes a `book.AAPL` snapshot — that bid level is now gone.
8. Clearing process receives `trade.executed` and updates both traders' P&L.

Result:

```
GW01: bought 100 AAPL @ 150.00 — position +100, avg cost 150.00
GW02: sold   100 AAPL @ 150.00 — position -100 (or flat if they were long)
```

2.7 Passive vs. aggressive orders

An order that **rests** on the book, waiting for a counterparty, is called **passive** (or a **maker** order, because it "makes" liquidity for others to trade against).

An order that **crosses** the spread and immediately matches against resting orders is called **aggressive** (or a **taker** order, because it "takes" available liquidity).

Role	Also called	What it does
Passive	Maker / resting	Posts to the book, waits
Aggressive	Taker / crossing	Matches immediately against the book

In real exchanges, makers often pay lower fees than takers as a reward for providing liquidity. EduMatcher does not model fees, but the distinction is important for understanding order strategy.

You don't have to infer maker/taker from order history, either: every `order.fill` event carries a `liquidity_flag` (`MAKER` or `TAKER`) stating it directly, on ALF, REST/WebSocket, and the drop-copy feed alike — see Your First Trade, Step 6 for it in action.

2.8 Key terms summary

Term	Definition
Order book	The live, sorted collection of all resting buy and sell orders for one instrument
Bid	A resting buy order; bids are sorted highest price first
Ask / offer	A resting sell order; asks are sorted lowest price first
Spread	Gap between best bid and best ask
Mid-price	Average of best bid and best ask
Depth	Total resting volume near the top of the book
Slippage	Difference between expected and actual average fill price on a large order
Price-time priority	Fill ordering rule: best price first, then earliest arrival
Passive / maker	An order resting on the book
Aggressive / taker	An order that crosses the spread and matches immediately

2.9 Implementation Note: Tick Prices

EduMatcher stores prices internally as integer ticks (not floating-point decimals). That means priority, comparisons, and level aggregation are exact in engine code. Displayed bid/ask values are converted back to decimals for user-facing views.

Glossary →

CHAPTER 3

A Full Trading Day

Learning objectives

After reading this page you will understand:

- How a real exchange trading day is structured into phases
- What happens during opening and closing auctions
- Which order types are valid in each phase
- How price discovery works at the open
- What happens to resting orders at phase transitions

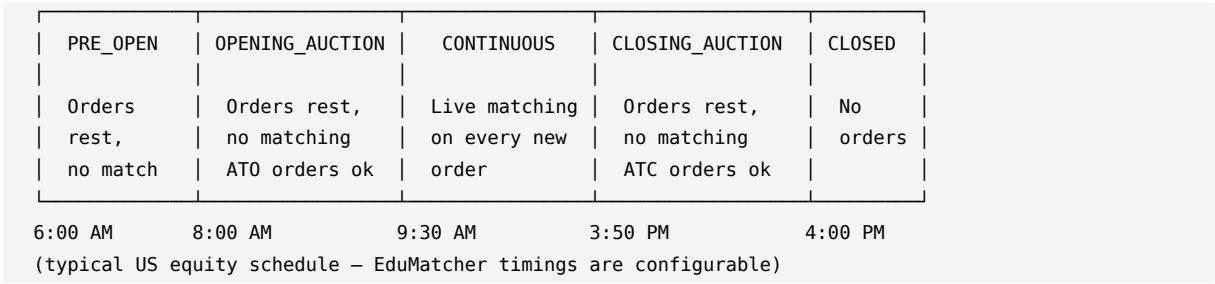
3.1 The shape of a trading day

A stock exchange does not run as a single continuous matching session from open to close. Instead, the day is divided into **session phases**, each with different rules about which orders are accepted and whether matching occurs.

This structure exists because:

- The opening price matters enormously (indices, ETF valuations, fund NAVs all reference it). A noisy first few seconds of continuous trading would produce a poor opening price.
- The closing price is even more important — it is the official daily settlement price used by clearinghouses and portfolio managers worldwide.
- Auctions give all participants equal footing at key moments, preventing fast players from exploiting slow ones at the open and close.

3.2 The five phases



Phase	Orders accepted?	Matching?	Special order types
`PRE_OPEN`	Yes	No	LIMIT, STOP, STOP_LIMIT, ICEBERG
`OPENING_AUCTION`	Yes	No	+ `TIF=ATO` (At-The-Open)
`CONTINUOUS`	Yes	Yes	All types including MARKET, FOK, IOC
`CLOSING_AUCTION`	Yes	No	+ `TIF=ATC` (At-The-Close)
`CLOSED`	No	No	— all orders rejected

3.3 Enabling or disabling session handling

Session-phase enforcement is controlled by `sessions_enabled` in `engine_config.yaml`:

```
sessions_enabled: true
```

- `sessions_enabled: true`: the engine starts in `CLOSED`, accepts new orders only in open phases, and enforces auction/session rules from scheduler transitions.
- `sessions_enabled: false`: the engine ignores `session.transition` messages and runs in perpetual `CONTINUOUS` behavior, which is useful for always-open local testing.

When sessions are enabled, starting the engine outside configured session times will keep it closed until `pm-scheduler` sends a transition to an order-accepting phase.

3.4 Specifying time for the daily session

Phase transition times are configured in `engine_config.yaml` under a `schedule` section, which is read by `pm-scheduler` (not by the engine itself). Each day of the week gets its own five-time block; `weekdays:` is a shortcut for Monday-Friday:

```
sessions_enabled: true

schedule:
  weekdays:
    pre_open:              "09:00"
    opening_auction_start: "09:25"
    continuous_start:      "09:30"
    closing_auction_start: "16:00"
    closing_auction_end:   "16:05"
```

All values are `HH:MM` in the **local time of the machine running the scheduler**.

Key	Phase triggered	Default (as `weekdays:`)
`pre_open`	`PRE_OPEN`	`09:00`
`opening_auction_start`	`OPENING_AUCTION`	`09:25`
`continuous_start`	`CONTINUOUS`	`09:30`
`closing_auction_start`	`CLOSING_AUCTION`	`16:00`
`closing_auction_end`	`CLOSED`	`16:05`

If the `schedule` section is absent entirely, the scheduler falls back to the built-in defaults shown above, applied Monday-Friday (weekends and bank holidays default to CLOSED). If a block such as `weekdays:` *is* present, it must define all five keys — a block missing one, e.g. `closing_auction_end`, is rejected when the config loads rather than silently skipping that one transition.

Saturday, Sunday and bank holidays can each be given their own block too (`weekend:`/`sat:`/`sun:`/`holidays:`) instead of defaulting to CLOSED — see Session Scheduling → Configuring the schedule for the full set of keys and worked examples.

`sessions_enabled` must be `true` for the scheduler-driven schedule to take effect. With it set to `false`, the engine ignores all `session.transition` messages and stays in perpetual continuous mode regardless of the `schedule` block.

3.5 Phase 1 — PRE_OPEN

The market is warming up. Traders can submit LIMIT orders that rest on the book, but no matching occurs. This allows participants to gauge where the market might open by watching the accumulating order book.

Example: GW01 submits a pre-open bid

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=200|PRICE=149.50
```

The order is accepted and rests. No trade happens yet.

3.6 Phase 2 — OPENING_AUCTION

The exchange accumulates all interest before computing a single opening price. The special `TIF=ATO` (At-The-Open) order type is only valid here.

Example: several participants add interest

```
# GW01 — institutional buy
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=500|PRICE=150.00|TIF=ATO

# GW02 — sell at limit
NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=300|PRICE=149.80|TIF=ATO

# GW03 — sell at limit (willing to sell at a lower price)
NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=400|PRICE=149.50|TIF=ATO
```

The accumulated order book before uncross might look like:

```
BID side (buys):          ASK side (sells):
200 @ 149.50 (pre-open)   300 @ 149.80
500 @ 150.00 (ATO)        400 @ 149.50
```

3.6.1 The uncross — how the opening price is computed

At the end of OPENING_AUCTION, the engine finds the price that **maximises the total quantity traded**. It does this by scanning all prices and asking: "if we execute at price P, how many shares would fill?" A buy order only counts as a "matchable buy" at a candidate price P if its own limit is P or higher — a buyer limited to $149.50 is not a willing buyer at $149.80.

For our example:

Candidate price	Matchable buys	Matchable sells	Exec qty	Surplus
149.50	200 + 500 = 700	400	400	300
149.80	500	300 + 400 = 700	500 ← maximum	200
150.00	500	300 + 400 = 700	500 (tied)	200

Above $149.50, GW01's pre-open 200-share bid drops out of "matchable buys" — its limit is $149.50, so it is not a willing buyer at $149.80 or $150.00 — leaving only GW01's 500-share ATO bid on the buy side. **Equilibrium price = $149.80**: both 149.80 and 150.00 reach the same maximum executable quantity (500 shares) and the same surplus (200), so the tie goes to the lower price. (The full tie-break rule minimises surplus first and only then prefers the lower price — see Session Scheduling → Equilibrium Price. Here surplus happens to be tied too, so the lower price decides directly.)

All crossable orders execute at $149.80: - GW01's ATO 500-share buy fills **in full** — the only buy order willing to pay $149.80 or more. - GW03's 400-share sell fills **in full** — its $149.50 limit is the better (lower) price, so it has priority over GW02's on the sell side. - GW02's 300-share sell fills **partially**: 100 of 300, which is all that is left of the 500 shares GW01's buy demands; the remaining 200 keep resting. - GW01's pre-open 200-share buy does **not** fill — its $149.50 limit is below the $149.80 clearing price — and keeps resting into CONTINUOUS.

Result published:

```
auction.result.AAPL: price=149.80 volume=500
```

Icebergs in the uncross

An ICEBERG's **hidden** quantity takes part in the uncross, not just its visible peak: the matchable buys and sells above count each order's full remaining quantity, so the indicative published during the call phase is what the uncross prints. At the clearing price the hidden part still queues **behind** the visible orders at the same price — the uncross fills one peak at a time, and each refilled peak goes to the back of the queue, exactly as in continuous trading.
After the uncross, ATO orders that didn't fill are **expired** (not cancelled by the trader — automatically removed by the engine). The `order.expired` event identifies the reason as `Opening auction ended`. The market transitions to CONTINUOUS.

3.7 Phase 3 — CONTINUOUS

This is the main trading session. Every new order is immediately swept against resting liquidity. All ten order types are valid here.

Price-time priority is in full effect. See The Order Book for a detailed explanation.

Example: continuous trading in action

```
# GW01 posts a limit bid
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=149.90

# GW02 sweeps it with a market order
NEW|SYM=AAPL|SIDE=SELL|TYPE=MARKET|QTY=100

# Trade: 100 shares @ 149.90
```

STOP and TRAILING_STOP orders activate here when `last_trade_price` crosses their trigger levels. See Stop Trigger Logic for details.

3.8 Phase 4 — CLOSING_AUCTION

The closing auction works exactly like the opening auction, but for the end of the day. The `TIF=ATC` (At-The-Close) order type is only valid in this phase.

The closing price is particularly important because: - **Index rebalancing** — index funds must buy/sell to match new index weights at the close, generating predictable large order flow. - **Fund NAV calculation** — mutual funds value their portfolios at the official closing price. - **Options expiry** — many options settle based on the closing price.

Example: closing auction

```
# Large index rebalancing buy
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=1000|PRICE=152.00|TIF=ATC

# Institutional sell (taking profit)
NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=800|PRICE=151.50|TIF=ATC
```

The uncross algorithm runs and produces the official closing price.

3.9 Phase 5 — CLOSED

The market is closed. All incoming orders are rejected:

```
[GW01] order.ack: accepted=false reason="Market is CLOSED"
```

When the engine shuts down it: 1. Serialises all resting **GTC** (Good-Till-Cancelled) orders to `data/gtc_orders.js` 2. Publishes `order.expired` for all resting **DAY** orders (they are discarded at close). 3. Publishes `system.eod` to all subscribers.

3.10 What happens to my orders at each transition?

Order TIF	At CONTINUOUS start	At CLOSED
`DAY`	Keeps resting	Expired and discarded
`GTC`	Keeps resting	Persisted to file, reloaded next day
`ATO`	**Expired** if unfilled at OPENING_AUCTION end	—
`ATC`	—	**Expired** if unfilled at CLOSING_AUCTIO

3.11 Triggering phase transitions in EduMatcher

In production, a scheduler process drives phase transitions automatically:

```
poetry run pm-scheduler --now
```

`--now` starts immediately in PRE_OPEN and progresses through all phases. Transition times are configurable in `engine_config.yaml`.

You can also trigger transitions manually from the admin console during development:

```
SESSION|STATE=OPENING_AUCTION
SESSION|STATE=CONTINUOUS
SESSION|STATE=CLOSING_AUCTION
SESSION|STATE=CLOSED
```

3.12 Putting it all together — a full-day scenario

Time	Phase	Event
06:00	PRE_OPEN	Engine starts; GTC orders reload from file
06:00–08:00	PRE_OPEN	Participants submit limit orders; book fills but no trades
08:00	→ OPENING_AUCTION	ATO orders now accepted
08:00–09:30	OPENING_AUCTION	Book accumulates; price discovery happening
09:30	→ CONTINUOUS	**Uncross at equilibrium price**; ATO unfilled orders expired
09:30–15:50	CONTINUOUS	Live matching; all order types active; stops triggering
15:50	→ CLOSING_AUCTION	ATC orders now accepted; new MARKETs rejected
15:50–16:00	CLOSING_AUCTION	Final accumulation
16:00	→ CLOSED	**Uncross at closing price**; ATC orders expired; GTC orders saved; DAY orders expired

Glossary →

CHAPTER 4

Gateways and How You Connect

Learning objectives

After reading this page you will understand:

- What a gateway is and what role it plays in an exchange architecture
- Why real exchanges offer multiple gateway protocols and what the major industry-standard formats are
- How EduMatcher simplifies the real-world concepts of users, participants, and members — and what those concepts mean in production
- Why arrival order at the matching engine matters and how real exchanges are legally obligated to handle it fairly
- Which gateway processes EduMatcher provides, and where to read the operational guide for each one
- Why `pm-alf-console`, the interactive terminal used throughout this guide, is technically **not** a gateway, and where the real ALF gateway lives

Prerequisites: the Quick Start Guide's Your First Trade. Your participant ID and role are set by the operator in the exchange's configuration before you can connect; the Operator's Guide chapter The Configuration Workflow explains how.

4.1 What is a gateway?

An exchange **gateway** is the entry point through which external participants send orders and receive market data and execution reports. It translates the external message format (FIX, binary, proprietary) into the internal format the matching engine understands, authenticates the sender, applies pre-trade risk checks, and routes messages to the right destination.

The gateway is deliberately kept outside the matching engine. The engine's only job is to match orders; it must not be slowed down by format parsing, session management, or rate limiting. Separating these concerns also allows the exchange to offer multiple gateway protocols simultaneously — an HFT firm and a retail broker can both connect to the same engine while speaking completely different wire formats.

A defining trait of a gateway, in this sense, is that it terminates an external transport (typically a TCP listener) and accepts connections from processes it does not control, often running on other hosts. A process

that only talks to the engine's own internal message bus, and cannot be reached from outside that bus, is not a gateway by this definition — see `pm-alf-console` is not a gateway for a concrete example.

4.2 Industry-standard gateway protocols

Real exchanges offer a range of gateway types. Each targets a different client population:

Protocol	Type	Used by	Notes
FIX 4.2 / 4.4 / 5.0	Text (tag=value)	Brokers, buy-side OMS	The lingua franca of institutional order routing; every major venue supports it; verbose but universally understood
OUCH (Nasdaq)	Binary	HFT, proprietary traders	Ultra-low latency; fixed-length binary fields; single-digit microsecond round trips
ITCH (Nasdaq)	Binary (market data only)	Market data consumers	One-way feed; used for direct order book reconstruction at co-location
FAST / SBE	Binary	Market data consumers	Simple Binary Encoding; used by CME, Eurex for market data
BOE (CBOE/BATS)	Binary	HFT	Binary Order Entry; competes with OUCH
ETI (Eurex)	Binary	European derivatives traders	Enhanced Transaction Interface; supports complex derivatives workflows
Proprietary REST/WebSocket	Text / JSON	Retail, algorithmic	Used by crypto exchanges and some retail venues; easy to integrate

EduMatcher's order-entry gateway speaks a **FIX-inspired pipe-delimited text format** that we call **ALF** (**AL**most **F**IX): `NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00`. It borrows FIX's field=value concept but uses a simplified subset - no session layer, no checksums, no sequence numbers, and no standard FIX message set. A production FIX gateway would add all of these.

Formal protocol reference

This page explains the general gateway concept and lists what EduMatcher provides. The formal syntax and semantics of the ALF protocol are defined in ALF Protocol Reference, and a protocol-by-protocol comparison of ALF/BALF/CALF/RALF lives in External Protocols Overview.

4.3 EduMatcher's gateways

EduMatcher ships one gateway (or gateway-like process) per protocol family. Each binds its own TCP port, is configured from its own section of `engine_config.yaml`, and can be started and restarted independently of the matching engine and of every other gateway.

Protocol	Process	Direction	Purpose	Operational guide
ALF	`pm-alf-gwy`	Order entry (in)	Text order entry for external bots and remote clients, over a plain TCP socket	ALF TCP Gateway
BALF	`pm-balf-gwy`	Order entry (in)	Binary, low-latency order entry for programmatic clients	BALF TCP Gateway
CALF	`pm-md-gwy`	Market data (out)	Subscribe/unsubscribe market-data feed: order-book snapshots, trade prints, session-state changes	Market Data Feed (CALF)
RALF	`pm-ralf-gwy`	Post-trade (out)	Replayable audit feed of executed trades for clearing, drop-copy, and audit consumers	Post-Trade Dissemination (RALF)
DC1	`pm-dc-gwy`	Drop copy (out)	Relays the engine's internal drop-copy feed to plain TCP clients that cannot speak ZeroMQ	Drop-Copy TCP Gateway
LALF	`pm-log-srv`	Logging (in), LALF-PS (out)	Collects operational logs from every other `pm-*` process; also distributes them to live log viewers	Centralized Log Server

Protocol	Process	Direction	Purpose	Operational guide
REST/WebSocket	`pm-api-gwy`	Order entry + market data	HTTP/JSON and WebSocket interface for browser and API-native clients	API Gateway

pm-alf-console is not in this table

`pm-alf-console`, the interactive trader console used throughout this guide's examples, is deliberately absent from this list: it connects directly to the engine's internal ZeroMQ bus instead of terminating a TCP port, so it does not fit the definition of a gateway above. See The Trader Console (`pm-alf-console`) for what it is instead, and ALF TCP Gateway for the process that *is* the real ALF gateway.

Each protocol's configuration lives in a different part of `engine_config.yaml`:

- **ALF** — configured under `participants`; used by `pm-engine` to authenticate order-entry connections from `pm-alf-console` and `pm-alf-gwy`, as well as `pm-balf-gwy` (the gateway id used in the BALF configurations must exist under `participants`). Uses a pipe-delimited text format (`FIELD=VALUE|FIELD=VALUE`).
- **BALF** — configured under the top-level `balf_gateway` key; used by `pm-balf-gwy`. Uses fixed-width binary frames with sequence numbers and integer-scaled prices, targeting programmatic clients where text-parsing overhead is undesirable.
- **CALF** — configured under the top-level `market_data_gateway` key; used by `pm-md-gwy`. Provides a subscribe/unsubscribe market-data feed delivering order-book snapshots, trade prints, and session-state changes over a persistent TCP connection with sequence-based gap detection.
- **RALF** — configured under the top-level `post_trade_gateway` key; used by `pm-ralf-gwy`. Provides a replayable audit feed of all executed trades, including the original order details, over a persistent TCP connection with sequence-based gap detection.
- **DC1** — configured under the top-level `dc_gateway` key; used by `pm-dc-gwy`. Relays the engine's internal drop-copy feed to plain TCP clients that cannot speak ZeroMQ, using the lightweight DC1 text protocol.
- **LALF** — configured under the top-level `log_server` key; used by `pm-log-srv`. Collects operational `logging`-module output from every other `pm-*` process over a persistent TCP connection into a queryable SQLite database. The same key also configures **LALF-PS**, the ZeroMQ `PUB`/`PULL` interface that distributes those rows back out to live log viewers.

See The Configuration Workflow for the full field-by-field schema of each gateway's configuration block, and External Protocols Overview for a protocol-centric comparison of ALF, BALF, CALF, and RALF.

4.4 One user per gateway — a learning simplification

EduMatcher maps one gateway process to one user. In a real exchange, the relationship between gateways, users, and legal entities has several layers:

```
Exchange
  └─ Member firm  (legal entity; signed exchange rules; financial responsibility)
       ├─ Participant  (trading desk or system within the firm)
       │    ├─ User  (individual trader or algorithm)
       │    └─ User
       └─ Participant
            └─ User
```

A single FIX session (one TCP connection to the exchange) can carry orders for many users in the same firm, tagged with a `SenderSubID` or `Account` field to identify the individual. Risk limits may be set at the firm level, the desk level, or the individual user level. Pre-trade checks (position limits, fat-finger checks, credit checks) can be applied independently at each layer.

EduMatcher collapses all of this:

- There is no concept of a member firm or legal entity.
- There is no concept of a "user" separate from the gateway.
- The gateway ID (`--id GW01`) is the only identity the engine knows.
- All orders from `GW01` are treated as one account for position tracking and self-match prevention purposes.

This makes the system much easier to learn and operate, at the cost of the access-control and risk-management structures that real venues require.

4.5 Multiple gateways and arrival order

When two gateways submit orders at almost the same moment, the engine processes them in the order the messages arrive at its PULL socket. On localhost with ZeroMQ, this is effectively FIFO — but only at the network level, not at the wall-clock level of the original submission.

In production this is a critical fairness issue:

- Two orders submitted at the same microsecond by two different participants on opposite sides of a co-location facility do not arrive at the engine at the same time.
- The order that traverses fewer network hops, or whose gateway server sits closer to the matching engine, will arrive first.
- This is the economics behind **co-location** services: participants pay to place their servers in the same data centre as the exchange, minimising the physical distance their messages travel.

Legal fairness obligations

Regulated exchanges are legally required to treat all participants fairly and without discrimination. In practice this means:

- **Deterministic FIFO processing**: the engine must process messages in the exact sequence they are received; it cannot re-order them for any reason.
- **No preferential access**: the exchange must offer the same co-location facilities and network connections to any participant willing to pay the published fee.
- **Timestamping**: many regulators (MiFID II in Europe, FINRA/SEC in the US) require the exchange to log a nanosecond-precision hardware timestamp ("gateway receipt timestamp") on every inbound message and include it in execution reports. This creates an auditable record of arrival order that can be reviewed by regulators after any suspicious trading pattern.
- **Speed bumps**: some venues (IEX, Cboe EDGA) deliberately introduce a short delay (350 microseconds for IEX's "Magic Shoebox") on certain order types to level the playing field between speed-optimised

HFT and slower participants.

EduMatcher has none of these mechanisms. The engine processes messages in ZeroMQ arrival order with no timestamps beyond the wall clock of the machine running the test. For a learning system on localhost this is irrelevant; for a regulated venue it would be a *compliance failure*.

4.6 See also

In this book:

- The Trader Console (`pm-alf-console`) — the learning/demo ALF client, and why it is not itself a gateway

Operator's Guide:

- ALF TCP Gateway — the real ALF gateway for external bots and remote clients
- BALF TCP Gateway
- Market Data Feed (CALF)
- Post-Trade Dissemination (RALF)
- Drop-Copy TCP Gateway (pm-dc-gwy)
- Centralized Log Server
- API Gateway
- Configuration — Participant Fields — how gateway IDs, roles, and disconnect behavior are configured

Protocols and Clients:

- External Protocols Overview — ALF/BALF/CALF/RALF compared side by side

CHAPTER 5

The Trader Console (`pm-alf-console`)

Learning objectives

After reading this page you will understand:

- Why `pm-alf-console` is an ALF client, not a gateway, and how that shapes its deployment (local machine only)
- How to start a session, submit every order type, manage positions, and read responses
- The full ALF command set: order entry, amendments, cancels, quotes, combos, OCO, monitoring, and session control

Prerequisites: Gateways and How You Connect — what a gateway is, why real exchanges have several, and why this process is not one. The Configuration Workflow — you need a valid `engine_config.yaml` with your gateway ID and role configured before connecting. Order Types comes right after this chapter, not before it — a genuine circular dependency: this chapter needs its command syntax, and it needs this chapter's console to try the examples. This chapter gives every field and example for `NEW`, `AMEND`, `OCO`, and `COMBO`; read Order Types alongside or straight after it for *why* each type behaves as it does (stop triggers, TIF lifetimes, combo risk). Messages — for the raw two-frame format underlying every response.

5.1 `pm-alf-console` is not a gateway

Despite sitting alongside the gateway chapters in this guide, `pm-alf-console` is not a gateway by the definition in Gateways and How You Connect: it does not terminate a TCP port and it cannot accept a connection from another process or another host. Instead, it is an **ALF client** that connects directly to the matching engine's internal ZeroMQ bus — a PUSH socket to the engine's PULL port and a SUB socket to the engine's PUB port — exactly like `pm-viewer`, `pm-audit`, or any other internal observer process.

Local machine only

`pm-alf-console` connects directly to the engine's ZMQ PUSH/SUB ports (`5555` / `5556`). It has no TCP listener of its own, so it **cannot accept connections from other hosts**. It is only suitable for demos, learning, and manual testing on the same machine as the engine.

For programmatic clients or any client running on a **different host**, use `pm-alf-gwy` instead. That process is the real ALF *gateway*: it binds a TCP port (default `5565`), accepts multiple ALF connections over the network, and bridges them to the engine's ZMQ bus — without any client needing ZMQ access.

The rest of this page documents `pm-alf-console` as a trading client: what it does, how to start it, its full command set, and how to read its responses.

5.2 What this process does

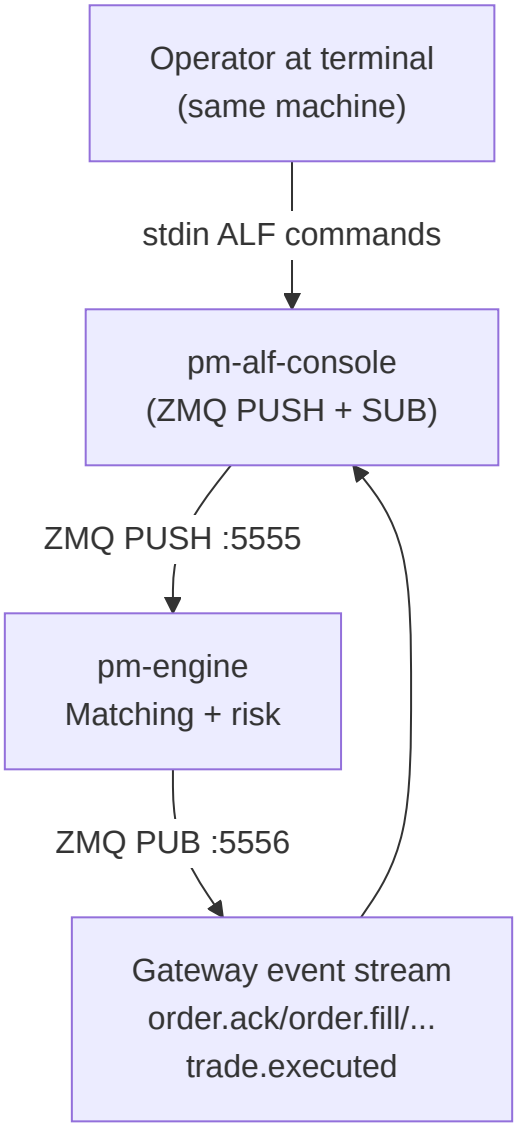
`pm-alf-console` is the **trader console**, the interactive ALF terminal, for local, in-session use. It is designed for humans sitting at the same machine as the running engine: it connects **directly to the engine's ZMQ sockets**, reads commands from stdin, and prints responses to stdout.

Responsibilities of `pm-alf-console`:

- authenticates the gateway ID against `participants`
- parses ALF command lines into validated engine requests
- subscribes to gateway-scoped lifecycle events (acks, fills, cancels, rejects)
- maintains local session caches for `ORDERS`, `POS`, `QBOOT`, and `QLEGS`
- provides interactive terminal ergonomics (history, tab completion, live events)

5.3 Architecture position

`pm-alf-console` sits **directly on the ZMQ bus**. It holds a ZMQ PUSH socket (sends commands to the engine) and a ZMQ SUB socket (receives engine events). There is no TCP layer between the operator and the engine — commands flow directly over ZeroMQ, which is why the process must run on the same host or at least have direct network access to the engine's ZMQ ports.



For clients on **another host**, `pm-alf-gwy` — the real ALF gateway — adds a TCP accept loop in front of the same ZMQ bridge, so the client only needs a plain TCP socket:



See ALF TCP Gateway for full documentation of `pm-alf-gwy`.

5.4 When to use ALF — protocol comparison

EduMatcher offers multiple interfaces. Use this quick guide to choose the right one.

Interface	Transport	Best for	Not suitable for
ALF (`pm-alf-console`)	ZMQ direct (local only)	Human/manual trading at the same machine — demos, learning, ad-hoc testing	Remote clients; any process on a different host
ALF (`pm-alf-gwy`)	TCP text `:5565`	External bots, remote scripts, any language, any host	Interactive terminal workflows (no tab completion, no P&L display)
CALF (`pm-md-gwy`)	TCP text	External market data (`TOP`, `TRADE`, `STATE`, `INDEX`)	Order entry
RALF (`pm-ralf-gwy`)	TCP text	External post-trade feeds (`CLEARING`, `DROP_COPY`, `AUDIT`)	Pre-trade market data, order entry
REST / WebSocket (`pm-api-gwy`)	HTTP/JSON + WS	Browser and API-native apps	Lowest-latency interactive trading
Internal ZMQ PUB/PULL	ZMQ binary	Internal processes that import `edumatcher`	External clients

Rule of thumb:

- Manual command-line order entry, **same machine** → **`pm-alf-console`**
- Automated bot or **remote host** → **`pm-alf-gwy`**
- External market data feed → **CALF**
- External clearing/audit feed → **RALF**
- Browser/web integration → **REST/WebSocket**

The interactive terminal (`pm-alf-console`) is your trading console for demos and learning. Each instance represents one user connecting to the trading system. Multiple instances can run simultaneously.

5.5 Starting `pm-alf-console`

```
poetry run pm-alf-console --id GW01
```

The `--id` flag sets your gateway identifier. It appears on all orders and fills. The ID must be preconfigured in `engine_config.yaml` under `participants`.

CLI flags:

Flag	Default	Description
`--id GW_ID`	*(required)*	Unique gateway identifier (e.g. `GW01`, `ALICE`). Must be listed under `participants` in `engine_config.yaml`
`--drop-copy`	off	Enable the drop-copy relay on startup (equivalent to sending `DC|STATE=ON` immediately after connecting). Can also be toggled at runtime — see DC — Toggle Drop-Copy Relay
`--log-level`	`WARNING`	Explicit level: `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG`
`-v` / `--verbose`	off	Increase verbosity (`-v` → `INFO`, `-vv` → `DEBUG`)
`-q` / `--quiet`	off	Reduce output to warnings/errors
`--log-target`	`server` (auto-detected `pm-log-srv`)	Where operational logs go: `server`, `stdout`, or `file`
`--log-file PATH`	—	Operational log file path; required when `--log-target file`
`--log-failover-timeout SECONDS`	30	Grace window before falling back to a local log file once `pm-log-srv` becomes unreachable
`--version`	—	Print the installed version and exit
`-h` / `--help`	—	Print the argument reference and exit

On startup, the terminal:

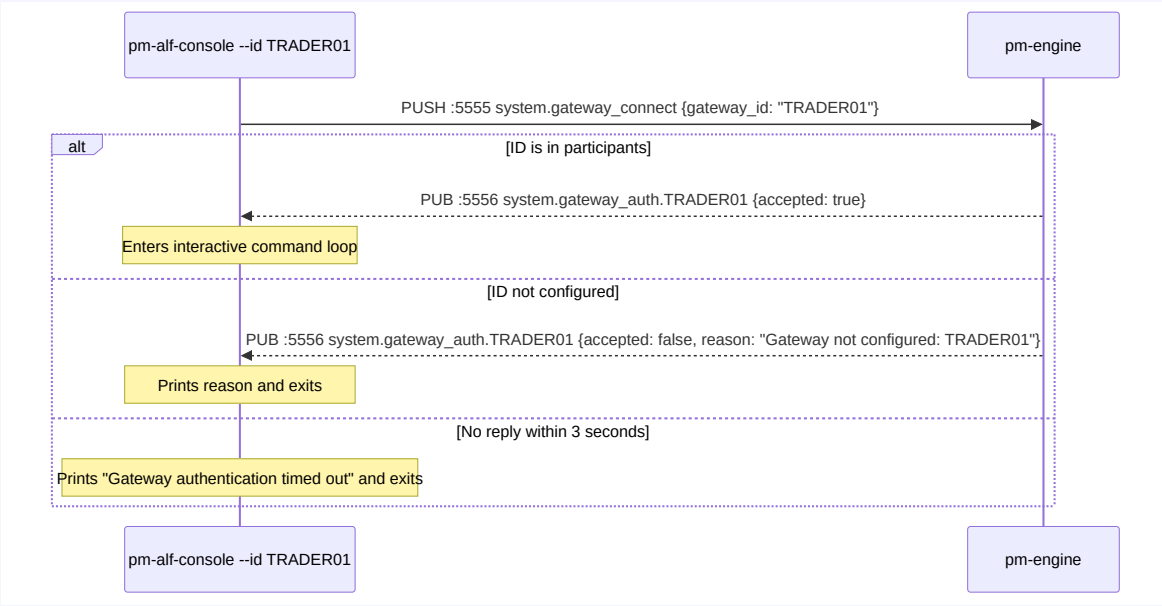
1. Connects PUSH socket to the engine PULL port (5555)
2. Connects SUB socket to the engine PUB port (5556)
3. Subscribes to: `order.ack.{ID}`, `order.fill.{ID}`, `order.amended.{ID}`, `order.cancelled.{ID}`, `order.expired` `order.orders.{ID}`, `combo.ack.{ID}`, `combo.cancel_ack.{ID}`, `combo.status.{ID}`, `oco.ack.{ID}`, `oco.cancel_` `oco.cancelled.{ID}`, `quote.ack.{ID}`, `quote.cancel_ack.{ID}`, `quote.status.{ID}`, `risk.kill_switch_ack.{I` `system.symbols.{ID}`, `system.quote_bootstrap.{ID}`, `system.gateway_auth.{ID}`, `trade.executed`
4. Sends `system.gateway_connect` and waits up to **3 seconds** for the auth response
5. If accepted: enters the interactive prompt loop
6. If rejected: prints the reason and exits immediately
7. If timeout (engine not running): exits with "Gateway authentication timed out"

A second pair of sockets talks to pm-index

Independently of the engine sockets above, `pm-alf-console` also opens a PUSH socket to the `pm-index` process's PULL port (`EDUMATCHER_INDEX_PULL_PORT`, default `5559`) and a SUB socket to its PUB port (`EDUMATCHER_INDEX_PUB_PORT`, default `5558`), subscribed to `index.update`, `index.history.{ID}`, and `index.error.{ID}`. These back the `INDEX` command family below and are independent of gateway authentication — `pm-index` does not need to be running for the rest of the console to work, but `INDEX` and `INDEX|HISTORY` will simply produce no output if it is not.

A third socket talks to the drop-copy feed

`pm-alf-console` also connects a SUB socket to the engine's drop-copy PUB port (`DROP_COPY_PUB_ADDR`, default `:5557`), but subscribes to nothing on it until `DC|STATE=ON` (or `--drop-copy` at startup) — see DC — Toggle Drop-Copy Relay below.



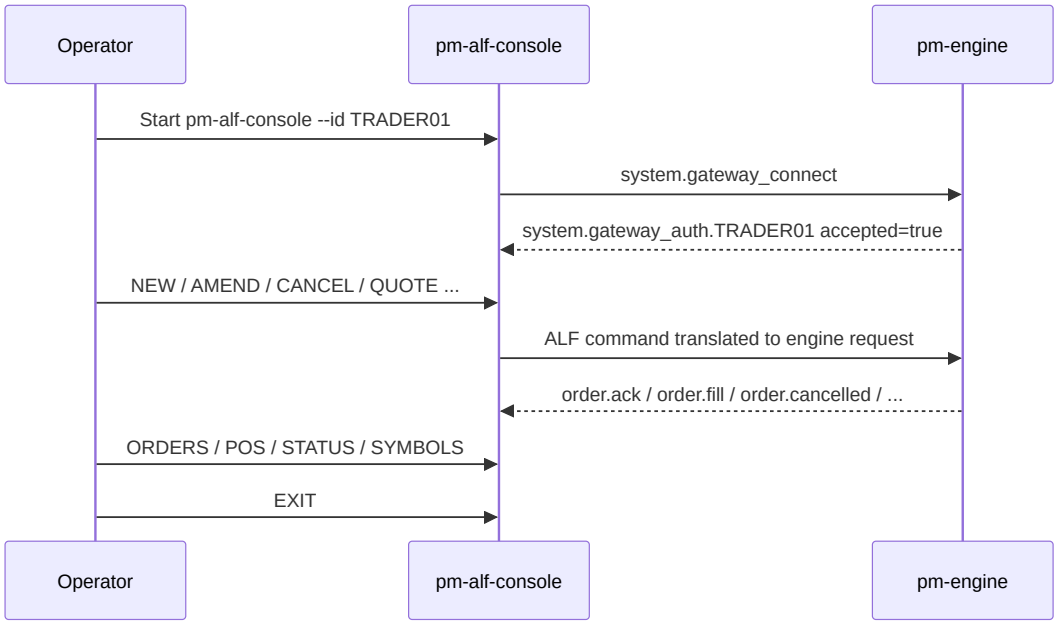
The terminal does **not** subscribe to `session.state`. Use `pm-audit`, `pm-viewer`, `pm-orders`, or the scheduler output if you need to watch trading phase transitions live.
Allowed gateway IDs are configured in `engine_config.yaml` under `participants`.
Example:

```
participants:
  - id: TRADER01
    description: The first trader
  - id: TRADER02
    description: High frequency
```

If a session starts with an ID that is not listed there, the engine refuses the connection and the terminal exits.

5.6 Session lifecycle

Every session follows this operator lifecycle:



5.7 Command Format

All commands use the ALF pipe-separated key=value format.

Note

For the precise ALF grammar, parser rules, field semantics, and full command catalog, see ALF Protocol Reference.

5.7.1 Command families at a glance

Family	Commands	Purpose
Order entry	`NEW`, `NEW|TYPE=COMBO`, `NEW|TYPE=OCO`, `QUOTE`	Create new exposure
Order updates	`AMEND`, `CANCEL`, `QUOTE_CANCEL`	Modify or remove exposure
Risk controls	`KILL`	Emergency local kill-switch
Drop copy	`DC`	Toggle asynchronous relay of this session's own fills from the engine's drop-copy feed
Monitoring	`STATUS`, `ORDERS`, `POS`, `SYMBOLS`, `SESSION`, `QBOOT`, `QLEGS`, `INDEX`	Inspect live/cached state
Session control	`HELP`, `EXIT`, `QUIT`	Terminal usability

5.7.2 Request tags for amend and cancel

TAG=<order-tag> is an optional client-supplied identifier on NEW. It identifies the order for its whole lifecycle and is echoed as TAG by ALF gateway clients, or as client_tag on engine, REST, and WebSocket payloads.

RTAG=<request-tag> is an optional client-supplied identifier on AMEND and single-order CANCEL. It identifies the **request**, not the order. The engine echoes it on the resulting AMENDED, CANCELLED, or rejected ACK, so a client can match an asynchronous response back to the exact amend or cancel it sent.

An order tag follows one order for its whole life, while RTAG lasts for one request. Use a different RTAG for each retry or concurrent amend/cancel against the same order. Engine-initiated cancels, such as OCO sibling cancels, combo cascades, kill switches, and session expiry, have no originating client request and therefore do not carry an RTAG.

5.7.3 Role entitlements (what each role can do)

Gateway role	Allowed trading behavior	Disallowed behavior
TRADER	Standard NEW/AMEND/CANCEL, OCO/COMBO, monitoring commands	MM quote commands (QUOTE, QUOTE_CANCEL)
MARKET_MAKER	All trader behavior plus quote workflows (QUOTE, QBOOT, QLEGS)	N/A within configured symbols/limits
ADMIN (if configured)	Operational commands per config/policy	Business flow outside policy

If a command is not allowed for your configured role, the terminal prints a rejection.

5.7.4 QUOTE — Submit/Replace A Two-Sided MM Quote

Tip

For automated quoting, see The Market-Maker Bot (pm-mm-bot).
QUOTE|SYM=|BID=|ASK=|BID_QTY=|ASK_QTY=[|TIF=<DAY|GTC>][|QUOTE_ID=]
Example:

```
[MM01]> QUOTE|SYM=AAPL|BID=209.80|ASK=210.20|BID_QTY=500|ASK_QTY=500|QUOTE_ID=Q1
[09:30:00.101] QUOTE ACK   Q1  bid=7c4a91e2 ask=be2170fd
[09:30:00.102] QUOTE ACTIVE  Q1
```

Rules:

- BID must be strictly less than ASK
- BID_QTY and ASK_QTY must be positive integers
- Existing quote for the same gateway+symbol is replaced

5.7.5 QUOTE_CANCEL — Cancel Active Quote

```
QUOTE_CANCEL|SYM=<symbol>
```

5.7.6 QLEGS — Inspect MM Quote Legs and Fill Flags

QLEGS prints a local quote-leg projection for the current session. It is intended for MARKET_MAKER ALF sessions where operators need a compact, low-cognitive-load view of which quote legs are still active and which have already traded.

pm-alf-console vs. pm-alf-gwy

This section describes pm-alf-console's QLEGS, which renders entirely from its own **local, session-scoped cache** built by observing this session's own quote/fill/cancel events — it never asks the engine. pm-alf-gwy also supports QLEGS with the same command syntax and column semantics described below, but forwards the request to the engine (system.quote_legs_request) and renders the engine's authoritative reply instead of a local cache — see ALF TCP Gateway → QLEGS. The RECENT/ALL history in both cases is real and current; it is kept in memory only and does not survive an engine restart — see Persistence → What is deliberately not persisted.

QLEGS[|SYM=][|SHOW=ACTIVE|RECENT|ALL]

Field	Required	Default	Description
SYM	No	all symbols	Restrict output to one symbol
SHOW	No	ACTIVE	ACTIVE = currently live legs, RECENT = completed legs, ALL = both

Output columns:

- Symbol, Quote, Leg (BUY/SELL), Order
- Price — the leg's limit price in display money, - when unknown
- Qty, Rem, Filled, Filled?
- Leg status, Quote status, Time

Filled? is YES/NO derived from Filled, not a separate fact; it exists so an operator scanning the table does not have to compare two numbers to answer the only question that usually matters.

Operator examples

1. Show only currently active quote legs across all symbols:

```
[MM01]> QLEGS
```

2. Show active legs for one symbol while managing a live quote:

```
[MM01]> QLEGS|SYM=AAPL
```

3. Show recent completed/cancelled legs to understand what just happened:

```
[MM01]> QLEGS|SHOW=RECENT
```

4. Full audit-style view (active + recent) for one symbol:

```
[MM01]> QLEGS|SYM=AAPL|SHOW=ALL
```

Example workflow (manual MM session)

```
[MM01]> QUOTE|SYM=AAPL|BID=209.80|ASK=210.20|BID_QTY=500|ASK_QTY=500|QUOTE_ID=Q123
[09:30:00.101] QUOTE ACK   Q123  bid=7c4a91e2 ask=be2170fd
[09:30:00.102] QUOTE ACTIVE  Q123

[MM01]> QLEGS|SYM=AAPL
# shows BUY leg 7c4a91e2 and SELL leg be2170fd as active, Filled?=NO

[09:31:02.417] FILL      7c4a91e2  qty=100 @209.8  remaining=400  [PARTIAL]

[MM01]> QLEGS|SYM=AAPL|SHOW=ALL
# BUY leg now shows Filled=100, Filled?=YES, status=PARTIAL
# SELL leg state reflects remaining quote lifecycle events
```

`QLEGS` is read-only. It does not send modify/cancel actions to the engine. Use `QUOTE`, `QUOTE_CANCEL`, or `KILL` for control actions.

5.7.7 QBOOT — Request Quote Bootstrap State

`QBOOT` asks the engine for the current active quote slot state for this session. It is intended for MM startup/reconnect workflows where quote legs may have been seeded by config before the terminal connected.

```
QBOOT[|SYM=<symbol>]
```

Field	Required	Default	Description
`SYM`	No	all symbols	Restrict bootstrap response to one symbol

Examples:

```
[MM_AAPL_01]> QBOOT
# prints all active quote bootstrap entries owned by MM_AAPL_01

[MM_AAPL_01]> QBOOT|SYM=AAPL
# prints only AAPL bootstrap quote state for MM_AAPL_01
```

Sample output:

```
       Quote bootstrap - MM_AAPL_01
┌────────┬───────┬────────┬────────┬────────┬───────┬───────┐
│ Symbol │ Quote │ State  │ Bid    │ Ask    │BidRem │AskRem │
├────────┼───────┼────────┼────────┼────────┼───────┼───────┤
│ AAPL   │ Q123  │ ACTIVE │ 209.80 │ 210.20 │  500  │  500  │
└────────┴───────┴────────┴────────┴────────┴───────┴───────┘
```

Typical startup use:

1. Connect terminal / bot as `MM_<SYMBOL>_01` (or matching seed owner).
2. Run `QBOOT|SYM=<symbol>`.
3. If exactly one healthy two-leg quote is returned, adopt it.
4. If state is missing/partial, cancel and re-issue.

5.7.8 INDEX — Show Live Index Level / Structural History

INDEX prints the most recent index level update received from pm-index. INDEX|HISTORY requests the structural/audit history (index creation, corporate actions, constituent adds, delistings) for one index.

```
INDEX
INDEX|HISTORY[|INDEX=<index_id>][|FROM=<date>][|TO=<date>]
```

Field	Required	Default	Description
INDEX	No	the last index ID seen on index.update	Index to query history for
FROM	No	30 days before now	Start of the history window
TO	No	now	End of the history window

Bare INDEX reads from an in-memory cache populated by the index.update feed and does not send a request to the engine or pm-index — if no index.update has been received yet, it prints No index data received yet. Is pm-index running?.

INDEX|HISTORY sends a request to pm-index and prints a **structural** history table (INIT, CORP_ACTION, ADD_CONSTITU DELIST records) — it is not a level/EOD time series. For level and EOD history, use pm-stats-cli index-daily / index-snapshots instead; see Statistics and Reporting.

If INDEX= is omitted and no index.update has been seen yet, INDEX|HISTORY prints INDEX|HISTORY requires INDEX=<id> or prior index.update. and sends nothing.

FROM is local midnight at the start of its date. Date-only TO includes the entire date, through the final nanosecond before the next local midnight; the same date can be used for both bounds to query one full day. Exact timestamp bounds on the history request are inclusive.

Example:

```
[TRADER01]> INDEX
[09:31:00.512] TECH100  4213.55 [green]+12.30 +0.29%[/green] [dim]O=4201.25 H=4220.10 L=4198.00[/dim] OPEN

[TRADER01]> INDEX|HISTORY|INDEX=TECH100|FROM=2026-06-01|TO=2026-07-01
# prints a "Index structural history" table of corporate-action-style records
```

5.7.9 KILL — Trigger Kill-Switch

```
KILL
KILL|SYM=<symbol>
```

KILL cancels active quote legs and non-quote resting orders for the gateway.

5.7.10 DC — Toggle Drop-Copy Relay

```
DC|STATE=ON
DC|STATE=OFF
```

DC subscribes (or unsubscribes) this session to the engine's drop-copy feed (DropCopyPublisher, ZMQ PUB :5557 — see Drop Copy), scoped to **this gateway's own fills only**. Once DC|STATE=ON is active, every subsequent fill for this gateway arrives asynchronously as a DC_FILL line, in addition to (not instead of) the usual FILL line driven by order.fill.{GW_ID} on the main event bus.

```
[GW01]> DC|STATE=ON
[dim]DC ON[/dim]
...
[09:31:02.417] DC_FILL   7c4a91e2  AAPL  qty=100 @150.05  [TAKER]  #42  (drop_copy.event.GW01)
```

Disabled by default; start with `--drop-copy` to enable automatically on connect instead of sending `DC|STATE=ON` manually every session:

```
pm-alf-console --id GW01 --drop-copy
```

Why both FILL and DC_FILL?

`FILL` (from `order.fill.{GW_ID}` on `:5556`) is the trading session's own fill notification and always arrives regardless of `DC` state. `DC_FILL` is a second, independent notification sourced from the engine's dedicated drop-copy feed (`:5557`) — the same feed a real exchange would deliver to a separate risk/back-office recipient. Both carry `LIQUIDITY`/`liquidity_flag` (`MAKER`/`TAKER`) — they must always agree, since both derive it from the same trade. Seeing both for the same fill (with different envelopes: `DC_FILL` carries the drop-copy `seq`, not the order-lifecycle `remaining`/`status` fields `FILL` carries) is expected and mirrors how a real participant's own trading session and their firm's drop-copy recipient receive independent copies of the same execution. `pm-alf-gwy` supports the identical `DC|STATE=ON`/`DC|STATE=OFF` command for external TCP clients — see ALF TCP Gateway → DC.

5.7.11 NEW — Submit an Order

```
NEW|SYM=<symbol>|SIDE=<BUY|SELL>|TYPE=<order-type>|QTY=<quantity>[|PRICE=<price>][|STOP=<price>][|TRAIL=<offset>]
```

SMP (Self Match Prevention) values: `NONE`, `CANCEL_AGGRESSOR`, `CANCEL_RESTING`, `CANCEL_BOTH`. SMP prevents you from accidentally trading against your own resting orders. If `SMP` is omitted, the engine falls back to the gateway's configured `participants[].smp_action` (or `NONE` if none is configured) rather than a fixed default — see Configuration — Participant Fields for the config field, and Risk Controls — Self-Match Prevention for a full explanation of how SMP is enforced during matching and how the per-order value and the gateway default interact.

ATO / ATC orders

The `ATO` (At-The-Open) and `ATC` (At-The-Close) TIF values are accepted by the engine during the appropriate auction phase and are offered by the terminal's tab completion alongside `DAY` and `GTC` (see Tab Completion below). These orders are only valid during `OPENING_AUCTION` and `CLOSING_AUCTION` phases respectively.

Examples

Order Type	Command
Market buy	`NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=100`
Limit sell	`NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=152.00`
GTC limit	`NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=200|PRICE=310.00|TIF=GTC`
Stop-loss	`NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP|QTY=100|STOP=148.00`
Stop-limit	`NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP_LIMIT|QTY=100|STOP=148.00|PRICE=147.50`

Order Type	Command
FOK	`NEW|SYM=AAPL|SIDE=BUY|TYPE=FOK|QTY=100|PRICE=150.00`
IOC	`NEW|SYM=AAPL|SIDE=BUY|TYPE=IOC|QTY=100|PRICE=150.00`
Iceberg	`NEW|SYM=AAPL|SIDE=BUY|TYPE=ICEBERG|QTY=1000|PRICE=150.00|VISIBLE=100`
Trailing stop	`NEW|SYM=AAPL|SIDE=SELL|TYPE=TRAILING_STOP|QTY=100|TRAIL=1.50`
With SMP	`NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00|SMP=CANCEL_RESTING`

Required fields by type

Type	Required fields	Optional
MARKET	SYM, SIDE, QTY	SMP
LIMIT	SYM, SIDE, QTY, PRICE	TIF, SMP
STOP	SYM, SIDE, QTY, STOP	TIF, SMP
STOP_LIMIT	SYM, SIDE, QTY, STOP, PRICE	TIF, SMP
FOK	SYM, SIDE, QTY, PRICE	SMP
IOC	SYM, SIDE, QTY, PRICE	SMP
ICEBERG	SYM, SIDE, QTY, PRICE, VISIBLE (must be < QTY)	TIF, SMP
TRAILING_STOP	SYM, SIDE, QTY, TRAIL	STOP (initial stop price), TIF, SMP

5.7.12 NEW (Combo) — Submit a Multi-Leg Order

```
NEW|TYPE=COMBO|COMBO_ID=<label>|COMBO_TYPE=AON|TIF=<DAY|GTC>|LEG_COUNT=<n>|LEG0.SYM=<sym>|LEG0.SIDE=<BUY|SELL>|LE
```

Combo fields

Field	Required	Description
`TYPE=COMBO`	Yes	Signals multi-leg order
`COMBO_ID=<label>`	Yes	Your tracking label (used for cancel)
`COMBO_TYPE=AON`	No	All-or-none semantics; defaults to `AON`, currently the only supported value
`TIF=DAY|GTC`	No	Time-in-force (default DAY), applies to all legs
`LEG_COUNT=<n>`	Yes	Number of legs (2–10)
`LEG<i>.SYM`	Yes	Symbol for leg i (0-indexed)
`LEG<i>.SIDE`	Yes	BUY or SELL
`LEG<i>.QTY`	Yes	Quantity

Field	Required	Description
LEG<i>.PRICE	Yes*	Limit price (*required for LIMIT, FOK, STOP_LIMIT, ICEBERG leg types)
LEG<i>.TYPE	No	Order type (default LIMIT)
SMP=<action>	No	Self-match prevention, applied to every leg; same values as NEW's SMP. If omitted, falls back to the gateway's configured participants[].smp_action (else NONE) — see Configuration — Participant Fields

Examples

Strategy	Command
Pairs trade	NEW|TYPE=COMBO|COMBO_ID=PAIR-001|COMBO_TYPE=AO
3-leg arb	NEW|TYPE=COMBO|COMBO_ID=ARB-01|COMBO_TYPE=AON\

Constraints

- 2–10 legs per combo
- No duplicate symbols (each leg must be a different instrument)
- All legs validated against engine_config.yaml symbol allowlist

5.7.13 AMEND — Amend a Resting Order

```
AMEND|ID=<full-order-id>[|PRICE=<new-price>][|QTY=<new-total-qty>][|RTAG=<request-tag>]
```

At least one of PRICE= or QTY= must be present. Only resting LIMIT and ICEBERG orders can be amended — amending any other order type is rejected with Cannot amend <TYPE> orders.

Field	Required	Description
ID	Yes	Full order UUID (visible in the ORDERS table)
PRICE	Conditional	New limit price; omit to keep current price
QTY	Conditional	New total quantity; must exceed the already-filled quantity
RTAG	No	Request tag echoed on AMENDED or rejected ACK

Priority rules:

Change	Time priority
Quantity decrease only	**Preserved** — the order keeps its queue position
Price change	**Lost** — the order moves to the back of the queue at the new price
Quantity increase	**Lost** — the order moves to the back of the queue

Reply: `AMENDED <id> price=<p> qty=<q> remaining=<r> [(priority reset)] rtag=<request-tag>` on success (the `(priority reset)` suffix appears only when the amend cost the order its time priority; `rtag` appears only when `RTAG` was supplied), or a rejection via `REJECTED` with a reason.

5.7.14 NEW (OCO) — Submit a One-Cancels-Other Pair

An OCO pair links two orders on the same symbol so that when one fills or is cancelled, the engine automatically cancels the other.

```
NEW|TYPE=OCO|OCO_ID=<label>|SYM=<symbol>|QTY=<qty>[|TIF=<DAY|GTC>]
   |LEG1_SIDE=<BUY|SELL>|LEG1_TYPE=<type>[|LEG1_PRICE=<p>][|LEG1_STOP=<p>][|LEG1_TRAIL=<offset>]
   |LEG2_SIDE=<BUY|SELL>|LEG2_TYPE=<type>[|LEG2_PRICE=<p>][|LEG2_STOP=<p>][|LEG2_TRAIL=<offset>]
```

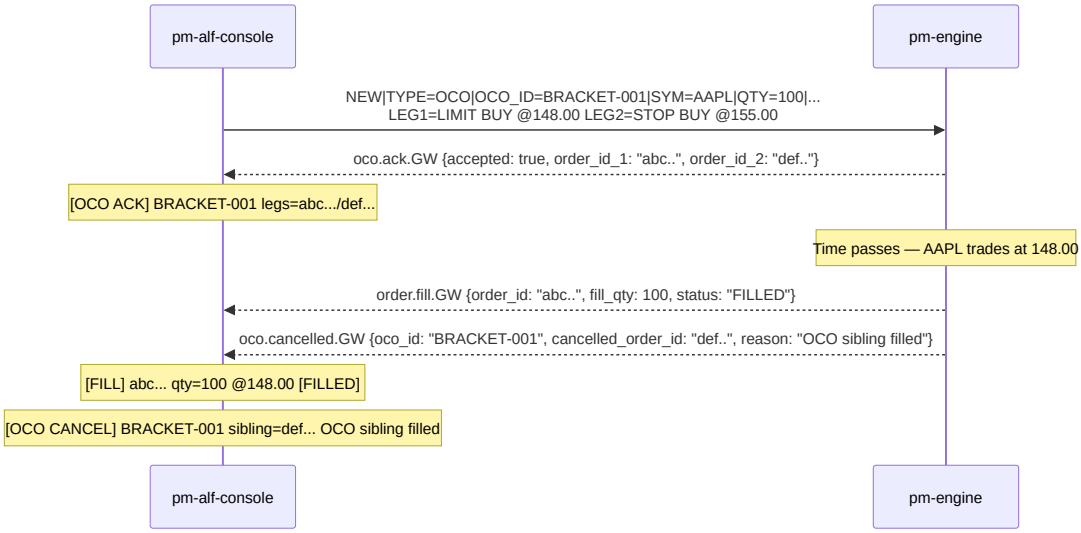
Field	Required	Description
`OCO_ID`	Yes	Client label for the pair
`SYM`	Yes	Instrument ticker — shared by both legs
`QTY`	Yes	Quantity — shared by both legs
`TIF`	No	`DAY` or `GTC`; defaults to `DAY`
`LEG1_SIDE`	Yes	`BUY` or `SELL`
`LEG1_TYPE`	Yes	Order type for leg 1
`LEG1_PRICE`	Conditional	Required for `LIMIT`, `IOC`, `FOK` legs (a `STOP_LIMIT` leg only requires `LEG1_STOP`, not `LEG1_PRICE`)
`LEG1_STOP`	Conditional	Required for `STOP`, `STOP_LIMIT` legs
`LEG1_TRAIL`	Conditional	Required for `TRAILING_STOP` legs
`LEG2_*`	Same rules as LEG1	Second leg fields

Example — bracket order

Buy limit below the market + stop-loss above (common bracket structure for a short position):

```
NEW|TYPE=OCO|OCO_ID=BRACKET-001|SYM=AAPL|QTY=100|TIF=GTC
   |LEG1_SIDE=BUY|LEG1_TYPE=LIMIT|LEG1_PRICE=148.00
   |LEG2_SIDE=BUY|LEG2_TYPE=STOP|LEG2_STOP=155.00
```

If the LIMIT leg fills at 148.00, the engine automatically cancels the STOP leg. If the STOP triggers at 155.00, the LIMIT leg is cancelled.



5.7.15 CANCEL — Cancel a Resting Order, Combo, or OCO

```
CANCEL|ID=<full-order-id>[|RTAG=<request-tag>]  # single-leg order
CANCEL|COMBO_ID=<combo-label>      # combo and all its resting legs
CANCEL|OCO_ID=<oco-label>          # both legs of an OCO pair
```

Every event line (`ACK`, `FILL`, `CANCELLED`, `AMENDED`, quote and OCO events) and the `ORDERS` table show the full 32-character order ID; copy it from either. In a narrow terminal the `ORDERS` table may shorten the ID column with … — widen the window, or copy the ID from the event line.

For a single-order cancel, `RTAG` is echoed on the `CANCELLED` event or rejected ACK. Group cancels are identified by `COMBO_ID` or `OCO_ID` and do not use `RTAG`.

Cancelling a combo or OCO is atomic: all resting child legs are cancelled, but fills that already occurred are not reversed.

5.7.16 STATUS — View Gateway Summary

`STATUS` prints a quick local summary for the current session.

```
STATUS
```

Use it when you want to confirm the gateway identity, known symbols, cached order counts by lifecycle state, cached quote legs, and position symbols without opening the full order table.

Order inspection is via ORDERS

`STATUS` is a summary command. For detailed order inspection — full order IDs, quantities, remaining quantity, price, TIF, and current status — use `ORDERS`.

5.7.17 ORDERS — Inspect This Gateway's Orders

```
ORDERS
```

Prints a rich table of all **single-leg** orders submitted in this session with full order ID, current status, remaining quantity, and last update time. This is the primary command for order inspection inside `pm-alf-console`.

Note

Combo child orders ARE resting orders like any other and do appear in the `ORDERS` table once accepted — there is nothing in the table to identify them as combo legs versus single-leg orders. Combo-level lifecycle (acceptance, partial/full match, cascade-cancel) is tracked separately via real-time `combo.ack` and `combo.status` messages printed as they arrive.

5.7.18 POS — View Current Positions

```
POS
```

Displays a position summary table showing net quantity, average entry cost, last trade price, unrealized P&L, and realized P&L for each symbol traded:

```
┌──────────────────────────────────────────────────────────────────────┐
│                            Positions                                 │
├──────────┬──────────┬──────────┬──────────┬──────────┬──────────────┤
│ Symbol   │ Net Qty  │ Avg Cost │  Last Px │ Unreal P&L │    Real P&L  │
├──────────┼──────────┼──────────┼──────────┼──────────┼──────────────┤
│ AAPL     │    +100  │   150.25 │   151.00 │    +75.00  │        +0.00 │
│ MSFT     │     -50  │   415.00 │   414.50 │    +25.00  │      +120.00 │
│ TSLA     │       0  │        – │        – │         –  │      +340.00 │
└──────────┴──────────┴──────────┴──────────┴──────────┴──────────────┘
```

How it works:

- Positions are accumulated locally from `order.fill` events received by this gateway
- Last price per symbol is tracked from the `trade.executed` feed (all trades, not just this gateway's)
- Unrealized P&L = (last price − avg cost) × net quantity
- Realized P&L is booked when reducing or closing a position (average cost method)
- Flat positions (net qty = 0) with non-zero realized P&L are shown dimmed
- Positions reset when the terminal disconnects (they are session-local)

Tip

Use `POS` after each fill to monitor your exposure without switching to `pm-orders`.

5.7.19 POS|GW=<gateway_id> — Query Another Gateway's Position

```
POS|GW=MM_AAPL_01
```

Unlike bare `POS` above, this does **not** use this session's local fill ledger. It asks the *engine* what the given `gateway_id` — typically a running `pm-mm-bot` instance, but any connected gateway works, including your own — is currently holding, via `system.position_request` / `system.position_snapshot.{gateway_id}`. The reply is rendered as a simpler table (the engine's ledger has no P&L, only net position and average cost):

```
┌──────────────────────────────────────┐
│      Position — MM_AAPL_01           │
├──────────┬──────────┬────────────────┤
│ Symbol   │ Net Qty  │ Avg Cost       │
├──────────┼──────────┼────────────────┤
│ AAPL     │     -300 │         150.05 │
└──────────┴──────────┴────────────────┘
```

A gateway with no open positions prints `<gateway_id>: flat (no open positions).` instead of an empty table.

How it works:

- The console sends `system.position_request` for the given `gateway_id` and dynamically subscribes to `system.position_snapshot.{gateway_id}` just for the duration of this one query, unsubscribing once the reply arrives — a mistyped or offline gateway_id does not leave a stray subscription behind.
- A second `POS|GW=` query before the first one's reply arrives replaces it (the earlier subscription is dropped).
- This works for *any* gateway_id known to the engine, not just MM bots — it is exactly the same query a `pm-mm-bot`'s own inventory-skewing logic effectively answers itself, just visible to an operator from outside. See Market-Maker Bot → Inventory skewing for the bot side of this.

5.7.20 SYMBOLS — List Active Instruments

```
SYMBOLS
```

Requests the list of all symbols that currently have an active order book in the engine. The terminal sends the request and the engine replies with the current instrument list plus any available `symbol_meta` fields, which are printed as a rich table:

```
┌──────────────────────────────────────────────────────────┐
│                    Active Instruments                    │
├─────┬──────────┬──────┬──────────────┬────────────┬──────────┤
│   # │ Symbol   │ Tick │ MM Enforced  │ Max Spread │ Min Qty  │
├─────┼──────────┼──────┼──────────────┼────────────┼──────────┤
│   1 │ AAPL     │ 0.01 │ YES          │         10 │      100 │
│   2 │ MSFT     │ 0.01 │ NO           │         10 │      100 │
│   3 │ TSLA     │ 0.01 │ —            │          — │        — │
└─────┴──────────┴──────┴──────────────┴────────────┴──────────┘
```

When metadata is available, the columns mean:

- `Tick`: symbol tick size derived from engine config
- `MM Enforced`: whether MM obligation rules are currently enforced for this gateway/symbol
- `Max Spread`: effective `mm_max_spread_ticks`
- `Min Qty`: effective `mm_min_qty`

Note

`SYMBOLS` returns symbols that have at least one active order book (created when the first order for a symbol arrives or when market-maker orders are injected at startup). It does NOT list all symbols from `engine_config.yaml` unless those symbols already have orders.

Gateway authorization

SYMBOLS and all trading commands are available only after successful startup authentication. If your ID is not listed under participants, the engine refuses the connection.

5.7.21 SESSION — Query Current Trading Session State

```
SESSION
```

Asks the engine for its current trading session state — PRE_OPEN, OPENING_AUCTION, CONTINUOUS, CLOSING_AUCTION, or CLOSED — right now, rather than waiting for the next unsolicited session-phase-change event. This is the fastest way to discover the current phase immediately after connecting or reconnecting.

```
[SC01]> SESSION
[09:31:02.104] SESSION  CONTINUOUS  (normal continuous matching)
```

If session-phase enforcement is disabled for this engine, an additional note is printed:

```
[SC01]> SESSION
[09:31:02.104] SESSION  CONTINUOUS  (normal continuous matching)
  [sessions gating disabled]
```

Note

Sending the SESSION command asks the engine for its current state on demand. The terminal does **not** subscribe to unsolicited session phase-change broadcasts — see the "Session phase changes" note under System Events below.

5.7.22 HELP — Command Reference

```
HELP
```

5.7.23 EXIT / QUIT — Disconnect

```
EXIT
```

5.8 Reconnect and state re-sync playbook

After restarting the terminal, rebuild local confidence in state with this sequence.

1. Run SYMBOLS to confirm active books and symbol metadata.
2. Run ORDERS to recover your current resting single-leg orders.
3. If you are a market maker, run QBOOT then QLEGS|SHOW=ALL.
4. Run POS to rebuild your session-local position view from current stream.
5. Resume normal trading only after steps 1-4 look consistent.

Why this matters:

- ORDERS gives authoritative order IDs for follow-up AMEND/CANCEL
- QBOOT/QLEGS avoids accidental duplicate quoting after reconnect
- POS helps detect drift between expected and observed exposure

5.9 Gateway Responses

All events are printed inline with a [HH:MM:SS.mmm] timestamp prefix. A background thread receives events from the engine while you continue typing.

5.9.1 Order Events

Message	Meaning
ACK <id> order accepted	Engine received and registered the order
REJECTED <id> code=<reject-code> rtag=<request-tag> <reason>	Order was rejected; code is stable for automation, rtag is present when the request supplied one
FILL <id> qty=50 @150.50 remaining=50 [PARTIAL]	Partial fill
FILL <id> qty=100 @150.50 remaining=0 [FILLED]	Full fill
AMENDED <id> price=151.0 qty=100 remaining=100	Amendment confirmed
CANCELLED <id>	Cancel confirmed
EXPIRED <id> (DAY order — trading day ended)	Engine shut down or session phase change

5.9.2 Quote Events

Message	Meaning
QUOTE ACK <quote_id> bid=<order-id> ask=<order-id>	Both quote legs accepted and posted to the book; full leg order IDs
QUOTE REJ <quote_id> <reason>	Quote rejected (e.g. "Quote requires bid_price < ask_price", missing gateway role)
QUOTE <status> <quote_id> [reason]	Quote lifecycle update — status is ACTIVE, INACTIVE_BID_FILLED, INACTIVE_ASK_FILLED or CANCELLED

5.9.3 OCO Events

Message	Meaning
OCO ACK <oco_id> legs=<leg1-id>/<leg2-id>	Both legs linked; full order IDs of the two legs
OCO REJ <oco_id> <reason>	OCO rejected (invalid legs, symbol mismatch, etc.)
OCO CANCEL <oco_id> sibling=<order-id> <reason>	Engine auto-cancelled the sibling leg after the other leg filled or was cancelled

5.9.4 Combo Events

Message	Meaning
`COMBO ACK <combo_id> combo accepted`	Combo validated, child orders posted to books
`COMBO REJ <combo_id> <reason>`	Combo failed validation (e.g. "Duplicate symbols in combo legs")
`COMBO STATUS <combo_id> PARTIALLY_MATCHED`	At least one leg has filled
`COMBO STATUS <combo_id> MATCHED`	All legs fully filled
`COMBO STATUS <combo_id> FAILED <reason>`	A leg was cancelled/expired; siblings cascade-cancelled
`COMBO STATUS <combo_id> CANCELLED`	User-initiated cancel completed

5.9.5 Risk Events

Message	Meaning
`KILL ACK orders=<n> quote_legs=<n>`	Kill-switch applied; counts show what was cancelled
`KILL REJ <reason>`	Kill-switch rejected (not expected in normal operation)

5.9.6 Drop-Copy Events

Only printed while `DC` is enabled (`DC|STATE=ON` or `--drop-copy`) — see DC — Toggle Drop-Copy Relay.

Message	Meaning
`DC ON` / `DC OFF`	Confirms the relay was toggled locally (printed immediately, no round trip to the engine)
`DC_FILL <id> <symbol> qty=<n> @<price> [<MAKER|TAKER>] #<seq>` `(drop_copy.event.<GW_ID>)`	One event per fill, sourced from the engine's drop-copy feed (`:5557`) rather than the main event bus — arrives in addition to the ordinary `FILL` line

5.9.7 System Events

Message	Meaning
`Active Instruments` table	Response to `SYMBOLS` command
`<INDEX_ID> <level> ...`	Response to `INDEX`, printed from the cached `index.update` feed
`Index structural history` table	Response to `INDEX|HISTORY`
`INDEX ERROR <reason>`	`pm-index` rejected an `INDEX|HISTORY` request

Session phase changes

The terminal does **not** subscribe to `session.state` events. Trading phase transitions are not displayed in the terminal. To monitor session phases, run `pm-audit`, `pm-viewer`, or `pm-orders` in a separate terminal.

5.10 End-to-end operator walkthrough

This transcript shows a realistic manual session from connect to disconnect.

```
# Start terminal
$ poetry run pm-alf-console --id TRADER01

[TRADER01]> SYMBOLS
# confirms AAPL/MSFT active

[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00|TAG=ORDER-001
[09:30:00.100] ACK  7c4a91e2  order accepted

[TRADER01]> ORDERS
# copy full UUID from table if you plan to amend/cancel

[TRADER01]> AMEND|ID=7c4a91e2-...|PRICE=150.10|RTAG=AMD-001
[09:30:01.500] AMENDED  7c4a91e2  price=150.1 qty=100 remaining=100 (priority reset) rtag=AMD-001

[09:30:02.200] FILL  7c4a91e2  qty=40 @150.10  remaining=60  [PARTIAL]

[TRADER01]> POS
# exposure now reflects partial fill

[TRADER01]> CANCEL|ID=7c4a91e2-...
[09:30:03.000] CANCELLED  7c4a91e2

[TRADER01]> EXIT
```

Recommended habits:

- Always use `ORDERS` to copy full IDs before `AMEND`/`CANCEL`
- Check `POS` after fills, not only at end of session
- If behavior looks inconsistent, run the re-sync playbook above

5.11 Multiple `pm-alf-console` Sessions

Run as many sessions as needed — each in its own terminal window:

```
# Terminal A
poetry run pm-alf-console --id TRADER01

# Terminal B
poetry run pm-alf-console --id TRADER02

# Terminal C
poetry run pm-alf-console --id TRADER03
```

Before starting a new gateway ID (for example TRADER03), add it to engine_config.yaml in participants and restart the engine.

Each session only receives events for its own orders. Use pm-orders to see all gateways' activity.

5.12 Interactive Features

5.12.1 Tab Completion

The terminal provides **context-aware tab completion**:

Position	Completions
First word	NEW, QUOTE, QUOTE_CANCEL, QBOOT, QLEGS, KILL, AMEND, CANCEL, STATUS, ORDERS, POS, SYMBOLS, INDEX, HELP, CLEAR, EXIT, QUIT
After NEW|	SYM=, SIDE=, TYPE=, QTY=, PRICE=, STOP=, TRAIL=, TIF=, VISIBLE=, SMP=
After NEW|TYPE=COMBO|	COMBO_ID=, COMBO_TYPE=, TIF=, LEG_COUNT=, plus LEG0.SYM=, LEG0.SIDE=, etc.
After NEW|TYPE=OCO|	OCO_ID=, SYM=, QTY=, TIF=, LEG1_SIDE=, LEG1_TYPE=, etc.
After AMEND|	ID=, PRICE=, QTY=, RTAG=
After CANCEL|	ID=, COMBO_ID=, OCO_ID=, RTAG=
After TYPE=	All order types: MARKET, LIMIT, STOP, STOP_LIMIT, FOK, IOC, ICEBERG, TRAILING_STOP, COMBO, OCO
After SIDE=	BUY, SELL
After TIF=	DAY, GTC, ATO, ATC
After SMP=	NONE, CANCEL_AGGRESSOR, CANCEL_RESTING, CANCEL_BOTH
After SYM=	Known symbols (populated from the last SYMBOLS reply)

Press **Tab** to cycle through suggestions.

5.12.2 Command History

Press ↑ / ↓ arrow keys to navigate through previously entered commands (in-memory, session-scoped — history is not persisted across restarts).

5.12.3 Background Event Display

A background thread listens on the SUB socket and prints events (fills, cancels, combo status changes, session transitions) in real-time, interleaved cleanly with the command prompt. You can continue typing while events arrive.

5.13 Common mistakes and fast triage

Symptom	Typical cause	Fast check	Action
`Gateway authentication timed out`	Engine is not running/reachable	Is `pm-engine` running in another terminal?	Start `pm-engine` first
`Gateway not configured: GW01`	Gateway ID not in `engine_config.yaml`	Check `participants` list	Add ID under `participants` and restart engine
`REJECTED ... code=UNKNOWN_SYMBOL`	Symbol unknown to engine config	Run `SYMBOLS`	Use listed symbols or add symbol to config
`REJECTED ... code=INSTRUMENT_HALTED` (or `CIRCUIT_BREAKER_ACTIVE`)	Circuit breaker/operator halt	Check audit/viewer output	Wait for resume or use admin controls
`REJECTED ... code=COLLAR_BREACH`	Price too far from reference	Compare to recent trade prices	Reprice closer to market
Order rests but does not fill	No crossing liquidity	Check opposite side activity	Wait or improve price
`Quotes are only allowed for MARKET_MAKER participants`	Role is `TRADER`	Verify gateway role in config	Change role to `MARKET_MAKER` if intended
`AMEND`/`CANCEL` fails with unknown ID	Short ID used instead of full UUID	Run `ORDERS` and inspect full ID	Retry with full order UUID

5.14 See also

In this book:

- Gateways and How You Connect — what a gateway is, and why `pm-alf-console` is not one
- Order Types — full semantics for every order type accepted by the terminal

Operator's Guide:

- ALF TCP Gateway — the real ALF gateway, for remote/external clients
- Configuration — gateway roles, allowlists, disconnect behaviour, and MM obligations
- Risk Controls — how the engine enforces collars, halts, and kill switches on gateway flow
- Running the Engine — how to start `pm-alf-console` and verify the connection
- Index — `pm-index` process, index composition, and the `index.update`/`index.history` feed behind the `INDEX` command
- Statistics and Reporting — `pm-stats-cli index-daily`/`index-snapshots` for level/EOD history (vs. `INDEX|HI` structural records)

Protocols and Clients:

- ALF Protocol Reference — formal ABNF grammar and field rules for the pipe-delimited syntax
- Messages — the ZeroMQ messages the terminal publishes and subscribes to

CHAPTER 6

Order Types

Learning objectives

After reading this page you will understand:

- What each of the eight order types does and when to use it
- The difference between aggressive (market/IOC/FOK) and passive (limit/iceberg) orders
- How stop-family orders trigger and what happens after trigger
- How time-in-force controls order lifetime across session phases
- How combo and OCO orders manage multi-leg strategy risk

Prerequisite: The Order Book — you should understand price-time priority before diving into order type mechanics.

Key point: priority is not keyed on the order's timestamp

Queue priority ("time priority" in price-time priority) is determined by the engine-assigned **`arrival_seq`**, not by the client-supplied `timestamp` field sent with the order. See Priority Rules in the Order Amendment section below for the full explanation.

EduMatcher supports **eight order types** plus two multi-leg constructs (COMBO and OCO), from the simplest market order to a COMBO that trades a strategy across several symbols all-or-none and an OCO that links two exits on one symbol.

6.1 Contents

Order type	Category	Key characteristic
MARKET	Aggressive	Immediate execution at best price; no price guarantee
LIMIT	Passive	Rests on book at specified price or better
STOP	Conditional	Dormant until trigger; then becomes MARKET
STOP_LIMIT	Conditional	Dormant until trigger; then becomes LIMIT
FOK	Aggressive	All-or-nothing; fill completely or cancel

Order type	Category	Key characteristic
IOC	Aggressive	Fill what you can immediately; cancel rest
ICEBERG	Passive	Large order with hidden reserve; visible peak only
TRAILING_STOP	Conditional	Dynamic stop that follows price by fixed offset
COMBO	Multi-leg	Multiple linked orders across symbols
OCO	Multi-leg	Two orders where fill/cancel of one cancels the other

6.1.1 Choosing an order type

Pick the type from what you are trying to achieve, then check the trade-off you are accepting. Each type has its own section below with worked examples and a *How traders use it* subsection.

I want to…	Use	What I give up
Get in or out *now*, whatever the price	MARKET	Price certainty (slippage)
Get in or out now, but never worse than a price	IOC	Guaranteed completion — the rest is cancelled
Trade a whole size or nothing	FOK	Partial fills
Name my price and wait	LIMIT	Execution certainty and queue position
Work a large size without showing it	ICEBERG	Speed; the replenished peak joins the back of the queue
Exit automatically if the market falls (or rises) to a level	STOP	Price certainty after the trigger
Same, but with a worst acceptable price	STOP_LIMIT	Execution certainty after the trigger
Protect a profit that keeps growing	TRAILING_STOP	A reversal of `TRAIL` size is needed before it acts
Trade a relationship between instruments	COMBO	Each leg must be priced; fills only complete together
Have exactly one of two exits happen	OCO	Both legs share one symbol and one quantity

6.1.2 Fields each type takes

The same field rule is applied by every entry point (`pm-alf-console`, `pm-alf-gwy`, BALF, REST) and again by the engine, so a mistake gets the same reject code however it arrives. A required field that is absent is rejected with `MISSING_FIELD`; a field the type does not use is *rejected* with `UNSUPPORTED_FIELD`, never silently dropped.

Type	Required	Optional	Notes
`MARKET`	—	—	Any `PRICE=` / `STOP=` is refused
`LIMIT`	`PRICE`	—	
`STOP`	`STOP`	—	`PRICE=` is refused — use `STOP_LIMIT` for a limit
`STOP_LIMIT`	`STOP`, `PRICE`	—	
`FOK`	`PRICE`	—	
`IOC`	`PRICE`	—	A priceless IOC is not supported; use `MARKET`
`ICEBERG`	`PRICE`, `VISIBLE`	—	`VISIBLE` must be smaller than `QTY`
`TRAILING_STOP`	`TRAIL`	`STOP`	Without `STOP=` the engine starts it from the last trade

`TIF=` (default `DAY`) is accepted on every type; see Time-in-Force. `SMP=` and `TAG=` are likewise common to all single orders.

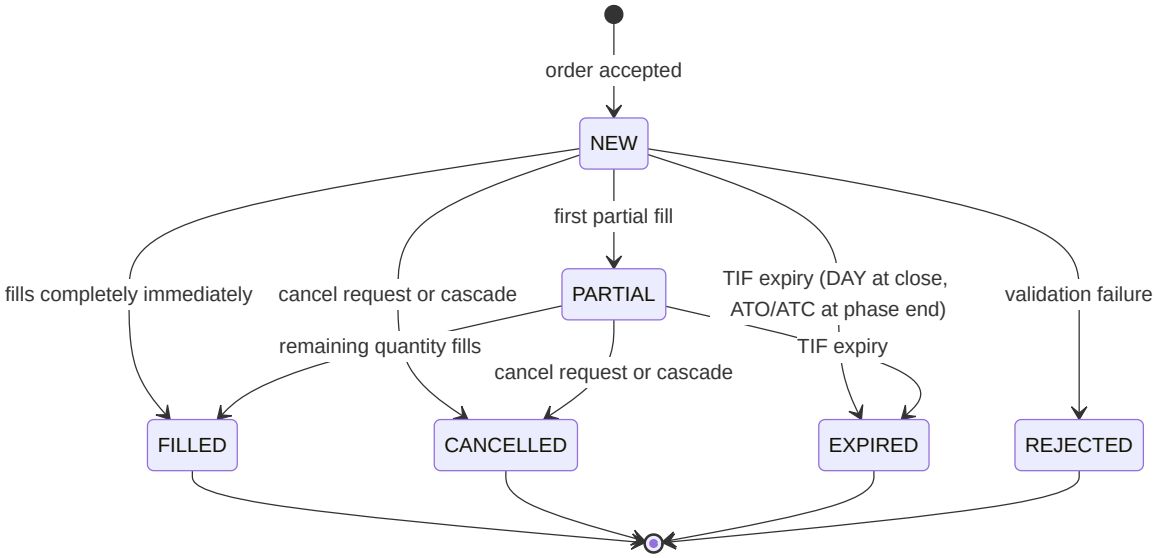
6.1.3 Behaviour by session phase

Type	Continuous trading	Auction / pre-open (no matching)	Symbol halted
`MARKET`, `IOC`, `FOK`	Match on arrival	**Rejected** `SESSION_NOT_PERMITTED` — they cannot rest	**Rejected** `INSTRUMENT_HALTED` / `CIRCUIT_BREAKER_ACTIVE`
`LIMIT`, `ICEBERG`	Match, then rest	Rest and are collected for the uncross (an iceberg's hidden quantity takes part, ranked behind its visible slice)	Rest without matching
`STOP`, `STOP_LIMIT`, `TRAILING_STOP`	Armed, or fired at once if already breached	Stored; evaluated after the uncross print	Stored

Everything is rejected with `MARKET_CLOSED` when the session is `CLOSED`.

6.1.4 Single-order lifecycle

Every order follows this state model from submission to final disposition:



6.2 MARKET

Back to Contents

Executes immediately at the best available price.

6.2.1 Background

Market orders are the oldest order instruction in listed markets. Before electronic books, brokers executed "at market" instructions on exchange floors by taking the best immediately available quote.

In modern matching engines, MARKET remains the purest urgency-driven order: it prioritizes immediacy over price certainty.

6.2.2 Motivation

Use MARKET when:

- Execution speed is more important than exact price.
- You are entering/exiting quickly in a liquid symbol.
- You are reducing directional risk and need certainty of fill.

6.2.3 How it behaves

- Sweeps the opposite side of the book (consumes resting orders from best price outward) as far as needed
- Partial fills are accepted — the unfilled remainder is **discarded** (not rested)
- Takes no `PRICE=`; supplying one is rejected (`UNSUPPORTED_FIELD`)
- Trades at each *resting* order's price, so a sweep produces one fill per price level touched and an average price worse than the best quote

The discarded remainder is not silently dropped. The order is *accepted*, and what could not be filled arrives as a cancellation carrying `CANCEL_REASON=INSUFFICIENT_LIQUIDITY` (`cancel_reason` on REST). A `MARKET` order

into an empty book is therefore an accepted order that immediately cancels in full — **not** a rejection, and worth distinguishing when you write a client:

Not shown in pm-alf-console

`pm-alf-console`'s terminal display prints `CANCELLED <order-id>` without the reason, even though the underlying message carries it. To see `CANCEL_REASON` for yourself, connect over the real ALF TCP gateway (`pm-alf-gwy`) with a raw client, or use the REST API's `cancel_reason` field — see ALF TCP Gateway — Unsolicited cancels.

Situation	What you receive
Book runs out during continuous trading	`ACK ACCEPTED=TRUE`, any `FILL`s, then `CANCELLED` with `INSUFFICIENT_LIQUIDITY`
Symbol halted, or the session is not matching	`ACK ACCEPTED=FALSE` with `INSTRUMENT_HALTED`, `CIRCUIT_BREAKER_ACTIVE` or `SESSION_NOT_PERMITTED` — rejected before the book is touched

`IOC` behaves identically for its own remainder. `FOK` differs: it is all-or-nothing, so an unfillable `FOK` is *rejected* with `INSUFFICIENT_LIQUIDITY` rather than accepted and cancelled.

6.2.4 Gateway Syntax

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=100
```

6.2.5 Example — a market buy walks the book

Asks: `50 @ 150.10`, `30 @ 150.20`, `100 @ 150.50`. You send the order above.

Fill	Price	Qty
1	150.10	50
2	150.20	30
3	150.50	20

Average price = $(50 \cdot 150.10 + 30 \cdot 150.20 + 20 \cdot 150.50) / 100 =$ **150.21**, i.e. 11 cents (the **slippage**) worse than the 150.10 you saw when you clicked. Had the book held only 80 shares, you would receive three fills for 80 and a `CANCELLED` for the other 20.

Educational note: Market orders **guarantee execution but not price**. In a thin book (few resting orders, so liquidity is low), a large market order can move the price significantly — this is called **slippage** (the difference between expected execution price and actual average fill price).

6.2.6 How traders use it

- **Emergency exits and hedges.** When the position is the problem (a news shock, a margin call, a hedge that must be on *before* the next print), the cost of waiting exceeds the cost of slippage.
- **Small size in a deep book.** If your quantity is well inside the top level, a market order costs one half-spread and nothing else.

- **Don't** send large market orders into a thin book or just after an announcement, when the spread is widest. Prefer an IOC with a limit a few ticks through the touch — a *protected market order* that takes what is there but refuses a runaway price.
- Market orders never rest, so they are the one type you cannot send during an auction call phase; use a marketable `LIMIT` there.

6.3 LIMIT

Back to Contents

Executes at the specified price or better; any unfilled portion **rests on the book** (remains posted as a passive order, waiting for a future counterparty to trade against it).

6.3.1 Background

LIMIT orders are the foundation of central limit order books (CLOBs). They define displayed liquidity and form the visible bid/ask ladder used by all other participants to price risk and route flow.

6.3.2 Motivation

Use LIMIT when:

- Price control is more important than immediate execution.
- You want to provide liquidity and potentially earn spread capture.
- You are comfortable waiting in queue under price-time priority.

6.3.3 How it behaves

- Buy LIMIT: fills at `price` or lower
- Sell LIMIT: fills at `price` or higher
- Becomes a resting order if not immediately matchable
- A *marketable* limit (one that crosses the book) trades first, at the resting orders' prices, then rests whatever is left at its own limit
- Subject to the symbol's price collar and tick size: a price too far from the reference, or off the tick grid, is rejected (`COLLAR_BREACH`, `TICK_VIOLATION`)
- Can be amended in place (the only type besides ICEBERG)

6.3.4 Gateway Syntax

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.50
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.50|TIF=GTC
```

6.3.5 Examples

Passive. Best bid 150.40, best ask 150.60. `BUY LIMIT 100 @ 150.50` does not cross, so it rests *inside* the spread as the new best bid. Everyone now sees 150.50 as the price to hit.

Marketable. Asks are `40 @ 150.60` and `100 @ 150.70`. `BUY LIMIT 100 @ 150.65` buys 40 @ 150.60 (price improvement over your limit), cannot touch 150.70, and rests the remaining 60 as a bid at 150.65.

Educational note: Limit orders **guarantee price but not execution**. In a thin book (few resting orders, so liquidity is low), a large limit order may not fill immediately or at all, especially if the limit price is far from the current market price.

6.3.6 How traders use it

- **Providing liquidity / capturing the spread.** Resting a bid and an offer around the mid earns the spread when both fill; this is what market makers do (see Market Making).
- **Queue position matters.** At one price, earlier `arrival_seq` fills first. Joining the best bid early beats improving the price by one tick late — but reducing quantity keeps your place, while a price change or a quantity increase sends you to the back (see Amendment).
- **Price discipline.** A buy limit under the market is a patient entry; a sell limit above the market is a take-profit target. With `TIF=GTC` it works across days until you cancel it.
- **Improving vs joining.** Stepping one tick ahead of the best bid buys priority at the cost of a tick; sitting at the touch costs nothing but time.

6.4 STOP (Stop-Market)

Back to Contents

A dormant order that **activates** when the market touches the stop price, then executes as a MARKET order.

6.4.1 Background

STOP orders became common as risk-management tools for both discretionary traders and portfolio overlays. They were designed to automate exits without requiring constant manual monitoring.

6.4.2 Motivation

Use STOP when:

- You need a hard, automated risk cutoff (stop-loss).
- You want to enter on momentum confirmation (breakout stop-entry).
- Guaranteed execution after trigger matters more than trigger-to-fill price precision.
- `STOP` price is the trigger, not the execution price
- Buy STOP: triggers when last trade price stop price (used to enter long on breakout)
- Sell STOP: triggers when last trade price $\leq$ stop price (used to limit downside loss)

6.4.3 Gateway Syntax

```
NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP|QTY=100|STOP=148.00
```

Takes `STOP=` only; `PRICE=` is rejected. While dormant the order is in no price level and appears in no depth, and it can be cancelled but not amended. Full trigger rules are in Stop Trigger Logic.

Educational note: Once triggered, a STOP becomes a MARKET order — execution price is not guaranteed, especially in fast markets.

6.4.4 Example — protective sell stop

You bought 100 AAPL at 150.00 and will accept a 2-dollar loss. Last trade is 150.30.

```
NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP|QTY=100|STOP=148.00|TIF=GTC
```

Nothing happens while trades print above 148.00. When a trade prints at 148.00 or lower, the order becomes `MARKET SELL 100` and hits the best bids. If bids are `60 @ 147.90` and `200 @ 147.50`, you sell 60 @ 147.90 and 40 @ 147.50 — an average of 147.74, below the 148.00 you chose.

If you place a sell stop at or above the last trade price (say `STOP=151.00` with the last trade at 150.30) it is **already breached and fires immediately** — a stop is not a resting limit order at that price.

6.4.5 How traders use it

- **Stop-loss.** The standard use: cap the loss on a long (sell stop below the market) or a short (buy stop above it) without watching the screen.
- **Breakout entry.** A buy stop above resistance enters only if the market proves it can trade there; see Why does a BUY STOP trigger at a price *higher* than the stop?.
- **Know the failure mode.** A stop is a *trigger*, not a price. In a gap or a thin book it fills far through the stop level, and a cluster of stops at one level can push the price into the next cluster (a cascade). Where that risk is unacceptable, use STOP_LIMIT; where the level should follow a rising market, use TRAILING_STOP.

6.5 STOP_LIMIT

Back to Contents

Like STOP, but once triggered converts to a **LIMIT order** at a specified limit price.

6.5.1 Background

STOP_LIMIT was introduced to address a classic STOP weakness: in fast or thin markets, trigger-to-fill slippage can be large. By adding a limit cap, traders can constrain worst-case execution price after trigger.

6.5.2 Motivation

Use STOP_LIMIT when:

- You need trigger automation but still require explicit price protection.
- You can tolerate non-fill risk after trigger.
- You want to avoid executing through temporary price spikes/gaps.
- Provides price protection after trigger, but may not fill if the market moves past the limit
- Requires both `STOP=` (trigger) and `PRICE=` (limit after trigger)

6.5.3 Gateway Syntax

```
NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP_LIMIT|QTY=100|STOP=148.00|PRICE=147.50
```

Educational note: The gap between stop price and limit price is the "insurance premium" against market impact. If the market falls through 147.50 before the order fills, it remains unexecuted as a resting limit.

6.5.4 Example — gap through the limit

Last trade 150.00. You hold 100 AAPL and place the sell above, `STOP=148.00`, `PRICE=147.50`. Overnight news moves the market and the next trade prints at 146.00 against bids that are all at 146.00 or lower.

- Triggered: 146.00 148.00, so the order becomes `LIMIT SELL 100 @ 147.50`.
- No bid is at 147.50 or higher, so nothing fills. The order **rests as the best offer at 147.50**, above the market, while the price keeps falling.

A plain STOP would have sold at about 146.00 and left you flat; the STOP_LIMIT protected the price and left you still long, in a falling market. If the market later rallies back to 147.50 the resting limit fills there.

6.5.5 How traders use it

- **Controlled exits in thin or gapping markets.** When a few ticks of slippage would defeat the point of the exit, the limit caps it.
- **Sizing the gap.** A tight gap (`STOP=148.00`, `PRICE=147.95`) is nearly a stop-market that may miss; a wide gap behaves like a STOP in orderly markets and only bites in a gap. Choose it from how far the instrument typically moves between prints.
- **Entries.** A buy stop-limit above resistance enters on the breakout but refuses to chase a spike beyond the limit.
- **Remember the leftover.** After the trigger it is an ordinary resting limit that must be watched, amended or cancelled. It stays at the back of the queue at that price (it receives a new queue position when it rests).

6.6 FOK (Fill-or-Kill)

Back to Contents

Must be **entirely filled immediately** or the whole order is cancelled.

6.6.1 Background

FOK comes from block-trading workflows where partial execution can break a strategy's economics. It is common in institutional routing logic and cross-venue execution algos that need all-or-nothing completion.

6.6.2 Motivation

Use FOK when:

- Partial fills are unacceptable.
- You need immediate certainty of complete size.
- The order is one leg of a tightly coupled strategy and partial execution introduces exposure.
- Checks total available quantity at the given price before executing

- The pre-check includes hidden iceberg reserve quantity (not just the displayed peak) and **excludes** same-gateway resting liquidity that self-match prevention would skip — so a FOK can reject even when the book's *displayed* total looks sufficient, if most of it belongs to your own gateway
- If `available_qty < order_qty` → rejected entirely (no partial fills ever)
- If sufficient → executes in full immediately

6.6.3 Gateway Syntax

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=FOK|QTY=100|PRICE=150.00
```

Educational note: FOK is used when partial fills are unacceptable — e.g. algorithmic strategies needing an exact block to execute a multi-leg trade atomically.

How a failed FOK is reported depends on why it failed:

Cause	Result
Not enough quantity at or better than `PRICE`	One `ACK ACCEPTED=FALSE, INSUFFICIENT_LIQUIDITY`; nothing traded
The shortfall is only because your own orders are in the way, and `SMP=CANCEL_AGGRESSOR` or `CANCEL_BOTH`	Accepted, then `CANCELLED` with `SELF_MATCH_PREVENTED`
Auction call phase	Rejected `SESSION_NOT_PERMITTED`

With `SMP=CANCEL_RESTING` or `CANCEL_BOTH` the same-gateway resting orders in the FOK's path are cancelled even when the FOK itself is then rejected for lack of liquidity — a FOK is not a free way to test a book you share with yourself.

6.6.4 Examples

Asks: `60 @ 150.00`, `50 @ 150.10`, `100 @ 150.30`.

- `BUY FOK 100 @ 150.10` — only 110 are available at or below 150.10, so the order fills as 60 @ 150.00 + 40 @ 150.10, in one go. **FILLED.**
- `BUY FOK 200 @ 150.10` — only 110 available. **Rejected**, no trade; the book is untouched.
- `BUY FOK 100 @ 150.00` — only 60 available at that price. **Rejected**, although more is offered one tick higher; widen `PRICE=` to include it.

6.6.5 How traders use it

- **Atomic block trades.** If 10,000 shares are needed for a hedge to work, eight thousand is not "most of the trade", it is a position you didn't want.
- **Strict liquidity test.** A FOK either gets the whole size at your price or tells you immediately, with no market footprint, that it isn't there — unlike an IOC, which takes whatever it can and signals your interest.
- **A leg that must not be partial.** When one leg's fill triggers another trade, partial execution leaves unhedged exposure.
- **Price it deliberately.** A FOK limited to the best ask only succeeds if the top level alone covers you. Setting `PRICE=` a few ticks through the touch lets it consume depth while still capping the price you will pay.

6.7 ICEBERG

Back to Contents

A large LIMIT order that hides most of its size, showing only a `visible_qty` "peak" at any one time.

6.7.1 Background

ICEBERG orders were designed for institutional execution where displaying full size can move the market against the trader. The displayed "tip" reveals only a small slice while preserving the same total parent quantity.

6.7.2 Motivation

Use ICEBERG when:

- You need to execute large notional size with reduced information leakage.
- You want to rest passively without broadcasting your full intent.
- You accept slower completion in exchange for lower signaling risk.
- Only `VISIBLE=` quantity is published to the order book
- Once the visible peak is fully consumed, a **new peak** is replenished from the hidden quantity
- Each new peak gets a **new `arrival_seq`** → goes to the back of the queue at that price level (because the book uses price-time priority: among orders at the same price, the order with the lowest `arrival_seq` — i.e. the one the engine admitted first — gets filled first; a fresh `arrival_seq` means this peak waits behind all other orders already resting at that price)
- The total hidden size is **never visible** to other market participants
- An **incoming** iceberg that crosses the book trades its **full** quantity immediately, like a LIMIT — one execution per resting counterparty, not one per peak. Only the remainder that rests is hidden: it rests with a full peak of `min(VISIBLE, remaining)`

Priority is keyed on arrival_seq, not the order's timestamp

`arrival_seq` is a monotonic counter the **engine** assigns when it admits an order into the book — it is not the client-supplied `timestamp` field on the order. See Priority Rules below and Order Amendment — AMEND for the full explanation of why this distinction matters.

6.7.3 Gateway Syntax

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=ICEBERG|QTY=1000|PRICE=150.00|VISIBLE=100
```

Educational note: This is how large institutional orders are worked into the market without telegraphing their full size. Watch the book viewer as the iceberg refills — the qty at that price level resets to 100 after each 100-lot fill. The order ID stays the same, but its `arrival_seq` is refreshed (new `arrival_seq` = new queue position — the order's original client `timestamp` does not change).

`VISIBLE` must be smaller than `QTY` (an iceberg showing everything is just a LIMIT) and both `PRICE=` and `VISIBLE=` are required.

6.7.4 Example — working 1,000 shares with a 100-share peak

Bids: `300 @ 150.00` (other traders, ahead of you in the queue).

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=ICEBERG|QTY=1000|PRICE=150.00|VISIBLE=100
```

1. **Rests with a 100 peak.** The book shows `400 @ 150.00` (300 + your 100). The other 900 are known only to the engine.
2. **Sellers arrive.** A `SELL 350 @ 150.00` fills the 300 ahead of you, then your 50 of the peak. Your peak now has 50 left showing.
3. **Peak exhausted.** Meanwhile another trader has bid 70 @ 150.00. A `SELL 50` takes your last 50. The peak is replenished to 100 shown, **at the back of the queue** — behind that 70 — and the book now shows `170 @ 150.00`, with 900 of your 1,000 still to trade.
4. Repeat until all 1,000 have traded. `FILL` messages report each execution against the same order ID.

If the same order had instead arrived while 150 shares were offered at 150.00, it would first trade those 150 in full — as any limit order does — and rest only the remaining 850, behind a fresh 100 peak.

An iceberg can be amended. A price change or a quantity increase refills the peak and loses priority; a quantity decrease keeps priority and can only shrink the current peak.

6.7.5 How traders use it

- **Reducing information leakage.** A visible 1,000-share bid invites others to trade ahead of you or to lean on the price. A 100-share peak looks like ordinary flow.
- **Choosing the peak.** Too small and your queue position resets constantly (each refill goes to the back) and a pattern of identical 100-lot refills reveals the iceberg; too large and you lose the point. Peaks comparable to the typical visible size at the level blend in best.
- **Cost of hiding.** You trade slower than a displayed order of the same size would: the hidden part has lowest priority at the price, and other participants' displayed orders get filled first.
- **In auctions** the hidden quantity counts toward the uncross, so an iceberg is not penalised in size terms for being in an opening or closing call.

6.8 COMBO

Back to Contents

A **multi-leg order** that bundles two or more child orders across different symbols into a single atomic instruction. The child orders are linked: they are submitted, tracked, and cancelled as a group.

6.8.1 Background

COMBO orders are the exchange-native answer to legging risk in spread, pairs, and relative-value strategies. Instead of managing legs manually, the matching system keeps lifecycle and cancellation coupling explicit.

6.8.2 Motivation

Use COMBO when:

- Strategy PnL depends on a relationship between instruments, not one symbol.
- You want coordinated child lifecycle and automatic cascade-cancel.

- You need operational safety against partial strategy completion.
- Each leg is posted to its own symbol's book as an ordinary order — a `LIMIT` unless `LEG<i>.TYPE=` says otherwise (see below)
- 2 to 10 legs, each on a **different** symbol
- All legs are tracked together under a parent combo ID
- If any leg is cancelled or expires, all remaining legs are automatically **cascade-cancelled** (unfilled quantities only — fills already executed are not reversed)
- The combo is considered MATCHED only when **all** legs are fully filled

Educational note: COMBO is not a synthetic construct built on top of regular orders — it is a first-class order type with native engine support for multi-leg lifecycle management and cascade cancellation.

6.8.3 Gateway Syntax

```
NEW|TYPE=COMBO|COMBO_ID=PAIR-001|COMBO_TYPE=AON|TIF=GTC|LEG_COUNT=2|LEG0.SYM=MSFT|LEG0.SIDE=BUY|LEG0.QTY=100|LEG0
```

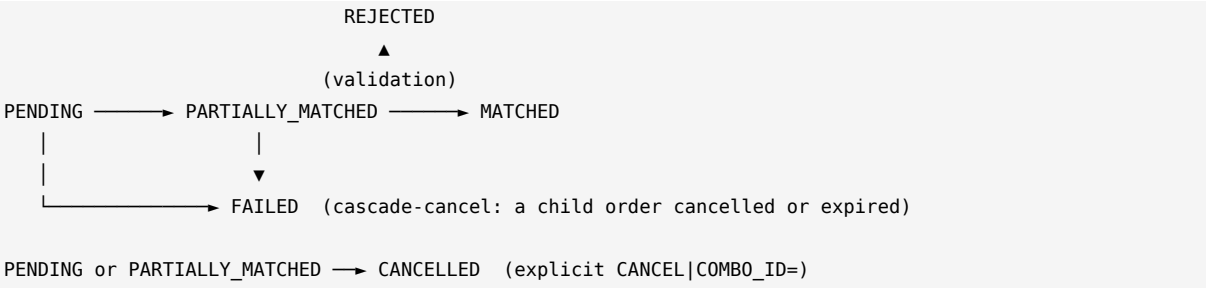
Field	Meaning
`TYPE=COMBO`	Identifies this as a multi-leg order
`COMBO_ID`	Your human-readable tracking label
`COMBO_TYPE=AON`	All-or-none: combo succeeds only when every leg fills completely
`LEG_COUNT`	Number of legs (2–10)
`LEG<i>.SYM`	Symbol for leg i
`LEG<i>.SIDE`	BUY or SELL for leg i
`LEG<i>.QTY`	Quantity for leg i
`LEG<i>.TYPE`	Optional; defaults to `LIMIT`. One of `LIMIT`, `MARKET`, `STOP`, `STOP_LIMIT`, `FOK`, `IOC`
`LEG<i>.PRICE`	Limit price for leg i (required for `LIMIT`, `FOK`, `IOC`, `STOP_LIMIT`)
`LEG<i>.STOP`	Trigger price for a `STOP` / `STOP_LIMIT` leg

`ICEBERG` and `TRAILING_STOP` cannot be combo legs (a leg has no `VISIBLE` or `TRAIL`); the combo is rejected with `INVALID_VALUE`. A combo's legs are collared like ordinary orders, and the whole combo is rejected if two legs share a symbol.

All-or-none in practice. Before posting any leg the engine checks that *every* leg could fill in full. If one is short and nothing would trade right now, all the legs rest as pending interest; if one is short but another leg could already trade, the combo is rejected (`INSUFFICIENT_LIQUIDITY`) with nothing posted, so you are never left holding half the structure.

Educational note: The `COMBO_TYPE` field is reserved for future expansion. Currently only `AON` (All-Or-Nothing) is supported, meaning the combo is successful only if every leg fills in full. In the future, we may add `ANY` (combo succeeds if at least one leg fills) or `MIN` (combo succeeds if a minimum number of legs fill).

6.8.4 Combo lifecycle



6.8.5 Cancelling a combo

```
CANCEL|COMBO_ID=PAIR-001
```

Cancels all unfilled legs. Fills already executed are irrevocable.

Educational note: Combos enable relative-value strategies (pairs trades, hedged entries, statistical arbitrage) where the profit depends on the *relationship* between instruments, not on any single instrument's direction. Without combos, submitting legs separately exposes you to **leg risk** — one side fills while the other does not, leaving you with unintended directional exposure. See the Combo Orders page for full details, examples, and the implied orders mechanism.

6.8.6 Example — a pairs trade

You think MSFT is rich relative to AAPL. You want to sell 100 MSFT at 415.00 and buy 200 AAPL at 210.00, and you want both or neither:

```
NEW|TYPE=COMBO|COMBO_ID=RV-1|COMBO_TYPE=AON|TIF=DAY|LEG_COUNT=2|LEG0.SYM=MSFT|LEG0.SIDE=SELL|LEG0.QTY=100|LEG0.PR
```

Submitted separately, a MSFT fill with no AAPL fill leaves you short one stock, outright, which is the risk you wanted to avoid. As a combo, the engine first checks that both legs could fill in full; if they can, both trade at once. If not, both legs rest. Either way, if one leg is cancelled or expires the other is cascade-cancelled, and the combo is `MATCHED` only when both have filled.

6.8.7 How traders use it

- **Pairs / relative value.** Profit comes from the spread between two names converging, not from either name's direction.
- **Hedged entries.** Buy the stock and sell the index or a related future in the same instruction, so you are never naked.
- **Calendar spreads.** Buy one expiry and sell another at a target difference.
- **Baskets.** Up to ten legs build a small basket in one order.
- **Know the limits.** When the legs cannot all fill at once they rest as separate orders in their own books, and one can trade before the other (status `PARTIALLY_MATCHED`). Price each leg so that the relationship is still worth having if only that leg fills for a while, and cancel the combo (`CANCEL|COMBO_ID=`) to withdraw the unfilled legs.

6.9 Time-in-Force (TIF)

Back to Contents

TIF (Time-in-Force) is not an order type; it is a set of instructions that specify how long an order remains active before it is executed or expires.

Time-in-force controls how long an order remains active before the engine automatically expires or cancels it.

6.9.1 Background

TIF instructions standardize order lifetime semantics across venues. They let participants express not only *how* to trade (order type) but *for how long* that instruction is valid under changing session states.

6.9.2 Motivation

Use TIF to align execution intent with horizon:

- `DAY` for session-local participation.
- `GTC` for persistence across restarts/sessions.
- `ATO`/`ATC` for auction-specific participation windows.
- `FOK` order type when immediacy and completeness must be coupled (note: `FOK` is an order type, not a `TIF=` value).

Value	Meaning
`DAY` (default)	Valid for the current trading session only. **Not** discarded at engine shutdown — it is persisted (alongside GTC orders) and restored on the next startup, dropped there only if the business day has since moved on. True expiry happens when the session transitions to `CLOSED`.
`GTC`	Good-Till-Cancelled. Persisted to `data/gtc_orders.json` at shutdown and reloaded next session. Survives across trading days.
`ATO`	At-The-Open. Only accepted during the `OPENING_AUCTION` phase. Automatically expired when the opening auction ends (transition to CONTINUOUS).
`ATC`	At-The-Close. Only accepted during the `CLOSING_AUCTION` phase. Automatically expired when the closing auction ends (transition to CLOSED).

FOK is an order type, not a TIF

`TIF=FOK` is **not valid** and will be rejected by the gateway. FOK semantics (fill entirely or cancel) are specified via `TYPE=FOK`, not `TIF=FOK`.
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00|TIF=GTC
The `ATO` / `ATC` window checks apply whenever trading sessions are configured; an order carrying one outside its window is rejected with `SESSION_NOT_PERMITTED`. They are meant for `LIMIT` (and `ICEBERG`) orders: a

`MARKET`, `IOC` or `FOK` cannot rest in an auction at all.

6.9.3 Examples

```
# Participate in the opening price, then disappear
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=500|PRICE=151.00|TIF=ATO
# Day-only resting bid
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=149.50
# Standing protective stop that survives the close
NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP|QTY=100|STOP=145.00|TIF=GTC
```

The first order is collected in the opening auction. If it fills at the uncross price, fine; any unfilled quantity is `EXPIRED` the moment the auction ends and never reaches continuous trading.

6.9.4 How traders use it

- **`DAY` for tactical orders.** Anything whose reason lapses at the close — a bid placed on today's news, a quote-chasing order — should expire with the session so it cannot trade tomorrow on stale information.
- **`GTC` for standing instructions.** Take-profit targets, protective stops and trailing stops should outlive the session; they are saved at shutdown and restored on the next start. You must cancel them yourself, and review them after corporate events or gaps.
- **`ATO` to trade the open.** A limit order priced generously for its side (a high bid, a low offer) ranks first when the uncross allocates volume and trades at the single equilibrium price — never worse than your limit, and often better. If the auction is imbalanced on your side, you may still be filled only in part.
- **`ATC` to hit the benchmark.** Funds and index trackers are measured against the closing price; sending to the closing auction gets that price with no market-impact race against the close.

6.10 Stop Trigger Logic

Back to Contents

Stop-family orders (`STOP`, `STOP_LIMIT`, and `TRAILING_STOP`) are **event-driven**: they do not trigger from top-of-book quotes, midpoint, or best bid/ask changes. They trigger from the symbol's **last executed trade price**.

6.10.1 Why last trade price?

Using last trade price aligns trigger events with actual executions, which reduces false triggers that can occur from ephemeral quote flickers. It also makes replay and audit deterministic: every trigger can be traced to a trade event in the execution stream.

6.10.2 Trigger Conditions

Stops are evaluated after every trade event:

Order Side	Trigger Condition
BUY STOP	`last_trade_price >= stop_price`
SELL STOP	`last_trade_price <= stop_price`

6.10.3 Exact evaluation rules

The rules below pin down the cases the overview leaves open. They hold for `STOP`, `STOP_LIMIT` and `TRAILING_STOP` alike.

- **On entry.** A stop whose condition is *already* met by the last trade price when it arrives triggers immediately instead of waiting for the next trade. (While the market is not matching — pre-open, an auction call, a halt — the entry check is skipped and the stop is simply stored; it is evaluated after the next trade, normally the uncross print.)
- **Once per incoming order.** Stops are evaluated after the incoming order has finished matching, against the price of its **last** execution — not after each individual print of a multi-level sweep. This matters for a trailing stop: it ratchets once, to the sweep's final price.
- **Several at once.** When one evaluation triggers several stops they execute one after another: triggered BUY stops first (lowest stop price first, then in arrival order), then SELL stops (highest stop price first, then in arrival order), then trailing stops in arrival order. Stops triggered by *those* executions queue behind them and run in turn. The same holds for the stops an auction's uncross print triggers.

Why does a BUY STOP trigger at a price *higher* than the stop?

It seems backwards — normally you want to buy *low*. The key is that a BUY STOP is not a value-buying tool; it is a *momentum confirmation* tool. The stop price acts as a breakout threshold: the trader is saying "I don't want to buy unless the market proves it can trade at or above this level."

Common use cases:

- **Breakout entry**: A stock has been consolidating below $150 (resistance). A trader places a BUY STOP at $150.01. If the price breaks through and prints at or above that level, it signals that sellers are exhausted and momentum is upward — the trader enters only after that confirmation.
- **Short-squeeze protection**: A trader is short and places a BUY STOP above the current price as an automated exit. If the market rises against their position past the stop, the buy triggers to close (cover) the short before losses grow larger.
- **Trend-following systems**: Algorithms that "buy strength" deliberately wait for a new high to be set before entering, treating the breakout level as the entry signal rather than a price they'd prefer to avoid.

The counterpart, SELL STOP (`last_trade_price <= stop_price`), is the more intuitive stop-loss: trigger a sell when the market falls to or through a floor price. BUY STOP is simply the mirror image applied to the upside.

6.10.4 Lifecycle After Trigger

1. The stop leaves its dormant container (`_buy_stops`, `_sell_stops`, or `_trailing_stops`).
2. The engine converts it:
   - `STOP` -> `MARKET`
   - `STOP_LIMIT` -> `LIMIT` (retains the original `PRICE=`)
   - `TRAILING_STOP` -> `MARKET` (using current ratcheted `stop_price` for trigger only)
3. The converted order is re-processed through normal matching logic.
4. It is treated as new from that moment: a `STOP_LIMIT` whose limit rests gets a fresh `arrival_seq` (queue position) when it does — it does not inherit any priority from when it was placed. A dormant stop has none, since it is not on the visible book.

This means a triggered stop behaves exactly like a newly submitted active order of its converted type.

6.10.5 Session-Phase Interaction

- In non-matching phases, stops can be accepted and stored.
- Trigger checks are tied to trade events, so without trades there are no triggers.
- On phase transitions that produce uncross trades, stop evaluations run after those trade events and can create cascades.

6.10.6 Trigger Cascades

A single aggressive trade can trigger multiple dormant stops:

1. Trade updates `last_trade_price`.
2. One or more stops become trigger-eligible.
3. Converted orders are injected and may execute immediately.
4. Those executions update `last_trade_price` again.
5. Additional stops may now trigger.

This cascade is expected behavior in volatile moves and is why stop-heavy books can accelerate directional momentum.

6.10.7 Worked Scenarios

Scenario A — SELL STOP trigger and slippage

Gateway message:

```
NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP|QTY=100|STOP=148.00
```

- Dormant order: `SELL STOP qty=100 stop=148`
- Last trade before event: `149.50`
- Next execution prints at `147.20`.

Trigger check: `147.20 <= 148` -> true, so order converts to MARKET SELL. Fill can occur below 148 depending on available bids (slippage).

Scenario B — STOP_LIMIT trigger without fill

Gateway message:

```
NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP_LIMIT|QTY=100|STOP=148.00|PRICE=147.50
```

- Dormant order: `SELL STOP_LIMIT qty=100 stop=148 limit=147.50`
- Trade prints at `147.80` (triggered)
- Best bid drops to `146.90` before order can match.

Order converts to LIMIT SELL at 147.50 and rests unfilled until a buyer meets that price.

Scenario C — Multiple stops fire on one print

Gateway messages (three separate dormant orders):

```
NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP|QTY=100|STOP=149.00
NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP|QTY=100|STOP=148.00
NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP|QTY=100|STOP=147.00
```

Dormant sell stops: 149, 148, 147.

If a trade prints at 147.00, all three satisfy `last_trade_price <= stop_price`. All convert in sequence and are processed as active orders.

6.10.8 Practical Notes

- Trigger is binary; `stop_price` is not an execution guarantee.
- STOP prioritizes fill certainty over price certainty.
- STOP_LIMIT prioritizes price control over fill certainty.
- TRAILING_STOP adds adaptive trigger movement but still converts to MARKET on trigger.

6.11 IOC (Immediate-Or-Cancel)

Back to Contents

6.11.1 Background

Immediate-Or-Cancel (IOC) is a time-in-force discipline that has been a fixture of electronic trading since the early days of ECNs (Electronic Communication Networks) in the 1990s. It was developed to give traders a way to "take what is available right now" without inadvertently posting a resting order that could fill much later at a stale price.

The Chicago Mercantile Exchange (CME), NYSE, and most modern exchanges support IOC natively. It is especially important for algorithmic and high-frequency traders who cannot afford the risk of an old order sitting on the book after a quote has moved.

6.11.2 Motivation

Use IOC when:

- You need to sweep available liquidity at or better than your limit price, but you are not willing to wait for the market to come to you.
- You want to avoid being "on the hook" for shares if the price moves away between submission and fill.
- You are executing a leg of a larger strategy and a partial fill is acceptable, but a resting order is not.

IOC vs LIMIT: A `LIMIT` order rests on the book if it does not fill immediately. An `IOC` order never rests — the unfilled portion is cancelled before control returns to the caller.

IOC vs FOK: `FOK` (Fill-Or-Kill) requires the *entire* quantity to fill immediately or it is rejected. `IOC` accepts partial fills, cancelling only what could not be filled.

6.11.3 How It Works

1. The engine receives the `IOC` order.
2. If the session is in a non-matching phase (auction), the order is **rejected** immediately — IOC cannot rest on the book.
3. During continuous trading, the engine sweeps the opposite side of the book, filling against resting orders at progressively worse prices up to the IOC limit price.
4. After the sweep completes (or if no liquidity was available), any remaining quantity is **cancelled** and a `CANCELLED` event is published.
5. The IOC order is never added to the resting order book.

Educational note: IOC is a pure liquidity-taking instruction. It is designed to consume whatever is available at or better than the limit price, but it does not wait for liquidity to appear. If the book is empty or all orders are above (for BUY) or below (for SELL) the limit, the IOC will simply cancel itself without any fills.

6.11.4 Examples

Example 1 — Full fill

Gateway message:

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=IOC|QTY=10|PRICE=100.00
```

Book state (asks):

```
Ask  100.00  qty=10
Ask  101.00  qty=10
```

Result: - Trade: 10 × 100.00 - IOC order: FILLED - No resting order added to the book

Example 2 — Partial fill, remainder cancelled

Gateway message:

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=IOC|QTY=10|PRICE=100.00
```

Book state (asks):

```
Ask  100.00  qty=5   ← only 5 available at limit price
Ask  101.00  qty=10  ← above IOC limit, not swept
```

Result: - Trade: 5 × 100.00 - IOC order: remaining 5 units **CANCELLED** - 5 asks at 101.00 remain on the book

Example 3 — No fill, entire order cancelled

Gateway message:

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=IOC|QTY=10|PRICE=100.00
```

Book state (asks): empty (or all asks above limit)

Result: - No trade - IOC order: all 10 units **CANCELLED** immediately

Example 4 — Multi-level sweep

Gateway message:

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=IOC|QTY=9|PRICE=101.00
```

Book state (asks):

```
Ask   99.00  qty=3
Ask  100.00  qty=3
Ask  101.00  qty=3
Ask  102.00  qty=3
```

Result: - Trade: 3 × 99.00, 3 × 100.00, 3 × 101.00 - IOC order: FILLED - 3 asks at 102.00 remain untouched

6.11.5 Gateway Syntax

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=IOC|QTY=100|PRICE=150.00
```

An IOC never rests, so `TIF=DAY` and `TIF=GTC` make no difference to it — it is always immediate-or-cancel by definition. `TIF=ATO` / `TIF=ATC` are *not* ignored, though: the session-window check still applies, and an IOC that carries one is rejected with `SESSION_NOT_PERMITTED` (in continuous trading because the window is closed, in the auction itself because IOC cannot rest). Leave `TIF=` off.

6.11.6 Edge Cases and Gotchas

Scenario	Behaviour
Submitted during opening/closing auction	**REJECTED** `SESSION_NOT_PERMITTED` — IOC cannot rest
Symbol halted	**REJECTED** `INSTRUMENT_HALTED` or `CIRCUIT_BREAKER_ACTIVE`
Self-match (same gateway on both sides)	SMP rule applies to the IOC fill attempt
Price limit not specified	**REJECTED** `MISSING_FIELD` — IOC requires a `PRICE=`
Unfilled remainder	`CANCELLED` with `CANCEL_REASON=INSUFFICIENT_LIQUIDITY` (the order itself was accepted)

6.11.7 How traders use it

- **Protected market order.** `BUY IOC 500 @ best_ask + 3 ticks` takes everything up to three ticks through the touch and refuses worse prices — a market order with a seat belt.
- **Taking a visible quote without leaving a footprint.** If a displayed offer is attractive but may be gone in a millisecond, an IOC at that price takes what is still there and leaves nothing for others to see or trade against.
- **Sweeping to a price cap.** To buy "up to 99.00" across several levels, one IOC at 99.00 is cleaner than repeated limit orders.
- **Slicing algorithms.** Execution algos fire an IOC per slice and size the next one from the fill; any residue is reported at once instead of lingering as stale interest.
- **Retry loop.** After a partial fill you decide what to do with the remainder — re-price an IOC, rest a LIMIT, or stop — rather than hoping a resting order catches up.
- **IOC vs FOK.** Choose IOC when a partial fill is useful and FOK when it is not.

6.12 TRAILING_STOP

Back to Contents

6.12.1 Background

A trailing stop is a stop order whose trigger price automatically moves in the trader's favour as the market moves favourably — but *never* moves against the trader. It was popularised by retail brokerages in the 1990s and 2000s as a way to "let profits run while cutting losses".

Modern exchange systems (CME, NASDAQ, CBOE) all support native trailing stops. They are especially useful for momentum strategies and position management when a trader wants to stay long/short for as long as the trend continues, but exit automatically once a reversal of a given magnitude occurs.

6.12.2 Motivation

Use a trailing stop when:

- You want to protect a profit without capping the upside. A regular stop is static — once set at, say, $98, it stays at $98 even if the stock rallies to $130. A trailing stop at $2 behind would move up to $128 automatically.
- You are managing a position and cannot monitor it continuously.
- You want a systematic, emotion-free exit strategy.

Trailing stop vs regular STOP: A regular stop has a fixed trigger price. A trailing stop has a *dynamic* trigger price that ratchets in one direction (up for SELL, down for BUY) with the market.

6.12.3 Gateway Syntax

```
NEW|SYM=AAPL|SIDE=SELL|TYPE=TRAILING_STOP|QTY=100|TRAIL=2.00|STOP=98.00
```

6.12.4 How It Works

The trailing stop maintains two values:

- **`trail_offset`** (`TRAIL=`): a fixed price distance from the market.
- **`stop_price`**: the current trigger level, updated after each trade.

SELL trailing stop (protecting a long position)

- Goal: trigger only if the price *falls* by `trail_offset` from its peak.
- After each trade at price P:
  - New candidate stop: $P - \text{trail_offset}$
  - **stop_price is updated only if the candidate is higher** (ratchet upward)
  - Trigger condition: $P \leq \text{stop_price}$

BUY trailing stop (protecting a short position)

- Goal: trigger only if the price *rises* by `trail_offset` from its trough.
- After each trade at price P:
  - New candidate stop: $P + \text{trail_offset}$
  - **stop_price is updated only if the candidate is lower** (ratchet downward)
  - Trigger condition: $P \geq \text{stop_price}$

On trigger

The trailing stop is removed from the pending list and converted to a **MARKET** order, which is then processed immediately in the same trade cycle.

Initial stop price

If `STOP=` is supplied, that is the initial `stop_price`. If omitted, the engine computes the initial stop from `last_trade_price ± trail_offset`. If there is no last trade and no explicit `STOP=`, the order is **rejected**.

6.12.5 Examples

Example 1 — SELL trailing stop: price rallies then drops

Gateway message:

```
NEW|SYM=AAPL|SIDE=SELL|TYPE=TRAILING_STOP|QTY=100|TRAIL=2.00|STOP=98.00
```

Setup: last trade price = 100. SELL trailing stop: `trail=2`, `stop=98`.

```
Trade at 105  →  stop ratchets up  →  stop = max(98, 105−2) = 103
Trade at 110  →  stop ratchets up  →  stop = max(103, 110−2) = 108
Trade at 102  →  102 $\leq$ 108?  YES  →  TRIGGERED → MARKET SELL
```

The trailing stop locked in a floor of 108 as the stock rallied to 110. When the stock dropped back to 102, the stop fired.

Example 2 — SELL trailing stop: stop never moves down

Gateway message:

```
NEW|SYM=AAPL|SIDE=SELL|TYPE=TRAILING_STOP|QTY=100|TRAIL=2.00|STOP=98.00
```

```
stop = 98, trail = 2, last_trade = 100

Trade at 106  →  stop = max(98, 104) = 104
Trade at 95   →  95 $\leq$ 104?  YES  →  TRIGGERED
```

Note that the trade at 95 did *not* move the stop down to 93: the candidate ($95 - 2 = 93$) is lower than the current stop, so the ratchet leaves it at 104, and 95 104 triggers. The market gapped from 106 to 95 without printing near 104, so the resulting MARKET SELL goes out well below the stop level — **slippage on the stop trigger**.

Example 3 — BUY trailing stop: price falls then rises

Gateway message:

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=TRAILING_STOP|QTY=100|TRAIL=3.00|STOP=53.00
```

Setup: last trade = 50. BUY trailing stop: `trail=3`, `stop=53`.

```
Trade at 45   →  stop ratchets down  →  stop = min(53, 45+3) = 48
Trade at 40   →  stop ratchets down  →  stop = min(48, 40+3) = 43
Trade at 44   →  44 ≥ 43?  YES  →  TRIGGERED → MARKET BUY
```

The stop followed the price down from 53 to 43. When the price started recovering and crossed 43, the buy was triggered.

Example 4 — Cancelling a trailing stop before trigger

```
CANCEL|ID=<order_id>|RTAG=trail-cxl-001
```

A trailing stop can be cancelled at any time before trigger using its `order_id`. The engine marks the order CANCELLED, publishes an `order.cancelled` event, and drops it from the pending trailing-stop list the next time the list is evaluated (lazy deletion). A trailing stop cannot be amended.

Example 5 — Trailing stop that starts from the last trade

```
NEW|SYM=AAPL|SIDE=SELL|TYPE=TRAILING_STOP|QTY=100|TRAIL=2.00
```

Last trade is 150.00 and no `STOP=` is given, so the engine sets the initial stop at $150.00 - 2.00 = \mathbf{148.00}$. Prints at 151, 153 and 152 move the stop to 149, 151 and leave it at 151; a print at 151 (or below) then fires a MARKET SELL. A trailing stop with no explicit `STOP=` and no prior trade in the symbol is rejected (`MISSING_FIELD`).

If you supply a `STOP=` that is already breached by the last trade (for a SELL, `STOP=` at or above it), the order fires on entry rather than waiting.

6.12.6 Gateway Syntax

```
# With explicit initial stop
NEW|SYM=AAPL|SIDE=SELL|TYPE=TRAILING_STOP|QTY=100|TRAIL=2.00|STOP=148.00

# Without explicit stop — engine computes from last trade price
NEW|SYM=AAPL|SIDE=SELL|TYPE=TRAILING_STOP|QTY=100|TRAIL=2.00
```

6.12.7 Edge Cases and Gotchas

Scenario	Behaviour
No `TRAIL=` supplied	**REJECTED**
No `STOP=` and no prior trade	**REJECTED** (engine cannot compute initial stop)
Submitted during auction phase	**Accepted** — rests until the auction uncroses, then evaluated on next trade
Large `trail_offset` relative to price	Stop may never trigger if the market doesn't move that far
Triggered during uncross	The resulting MARKET order participates in the uncross matching round
Multiple trailing stops on the same symbol	All are evaluated after every trade in O(t) time (t = number of trailing stops)
Several prints in one sweep	Evaluated once per incoming order, against the sweep's **last** price: the stop ratchets once, to that price
Amend	**Rejected** — cancel and re-enter instead

6.12.8 How traders use it

- **Riding a trend.** Hold a long while the move continues and exit automatically once it reverses by `TRAIL`. There is no profit target; the trade ends only on a pullback.

- **Calibrating `TRAIL`.** Too tight (a few ticks) and ordinary noise stops you out of a healthy trend; too wide and you give back much of the gain before it fires. A common rule of thumb is a multiple of the instrument's typical short-term range, so a volatile name gets a wider trail than a quiet one.
- **Buy-side mirror.** A BUY trailing stop above the market covers a short and follows the price *down*, locking in the profit of a falling market.
- **Combined with a target.** Paired with a take-profit LIMIT in an OCO it gives a bracket whose stop side tightens as the trade works: whichever of target or reversal comes first ends the position and cancels the other.
- **Limits of the tool.** It reacts only to *trades*, converts to MARKET (so gaps still slip), and, being a stop, is dormant during auctions.

6.13 Order Amendment (AMEND)

Back to Contents

6.13.1 Background

Order amendment (also called **modify** or **cancel/replace** in FIX protocol terminology) allows a trader to change the price or quantity of a resting order without explicitly cancelling and re-submitting it. This is a fundamental operation on every real exchange — CME, NYSE, Eurex, LSE, and NASDAQ all support in-place amendment.

In FIX Protocol, this is `MsgType=G` (Order Cancel/Replace Request). The key semantic distinction from a plain cancel+new is that an amendment **preserves the order ID** and, under certain conditions, **preserves time priority**.

6.13.2 Priority Rules

EduMatcher implements the same priority rules used by most lit exchanges:

Amendment Type	Priority
Quantity decrease only (price unchanged)	**Preserved** — order keeps its original `arrival_seq`
Price change (any direction)	**Lost** — order is assigned a new `arrival_seq` and moves to the back of the queue at the new price
Quantity increase (price unchanged)	**Lost** — order is assigned a new `arrival_seq` and moves to the back of the queue
Price change + quantity change	**Lost**

Rationale: A quantity decrease cannot disadvantage other participants at the same price level (there is less competition for incoming aggressor flow). Any other change either alters the priority ranking (price change) or grants the order more opportunity to fill without having competed for that position (quantity increase).

Priority is keyed on arrival_seq, not on the order's timestamp

It is tempting to think of price-time priority as being governed by the order's timestamp — and colloquially that is a fine mental model. But the field that actually determines queue position is the engine-assigned

`arrival_seq`, a monotonic counter incremented every time the engine admits an order (or a re-priced/re-queued order) into the book.
The order's `timestamp` field, by contrast, is **client-supplied** at submission time (see ALF Protocol and the message reference) — it records when the client says it built the order, and a back-dated or clock-skewed value **cannot** be used to jump the queue. Only the engine's own `arrival_seq`, assigned the moment the order (or amendment) is accepted, decides fill order among orders resting at the same price.
In outbound order snapshots (`order.orders.{GW_ID}`) and in the `AMENDED` event, watch `arrival_seq` — not `timestamp` — to see whether an order kept or lost its place in the queue.

6.13.3 AMEND vs. Cancel+New

Aspect	AMEND	Cancel + New
Order ID	Preserved	New ID assigned
Message count	1 inbound + 1 response	2 inbound + 2 responses
Priority on qty decrease	Preserved	**Lost** (new order gets current timestamp)
Atomicity	Guaranteed — no gap where the order is absent from the book	Non-atomic — brief window with no resting order
Fill risk during gap	None	Another order could fill the opposite side during the gap
Filled quantity tracking	Continuous — partial fills before amendment are preserved	Resets to zero on the new order
Gateway state	Single cache entry updated	Old entry cancelled + new entry created

When to use Cancel+New instead: If you need to change the order's side, symbol, order type, or TIF, you must cancel and resubmit — amendment only modifies price and/or quantity of an existing resting order.

6.13.4 Supported Order Types

Only **LIMIT** and **ICEBERG** orders can be amended. Other order types either cannot rest on the book (MARKET, IOC, FOK) or have trigger semantics that make in-place amendment ambiguous (STOP, STOP_LIMIT, TRAILING_STOP).

6.13.5 Gateway Syntax

```
# Change price only (priority lost)
AMEND|ID=<order-id>|PRICE=151.00|RTAG=amd-price-001

# Decrease quantity only (priority preserved)
AMEND|ID=<order-id>|QTY=50|RTAG=amd-down-001

# Change both price and quantity (priority lost)
AMEND|ID=<order-id>|PRICE=151.00|QTY=200|RTAG=amd-both-001
```

6.13.6 Examples

```
# 1. Place a resting buy limit
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00
# → ACK: order_id = abc123...

# 2. Market moves — improve price (priority lost, back of queue at 151)
AMEND|ID=abc123...|PRICE=151.00|RTAG=amd-price-002
# → AMENDED: price=151.0 qty=100 remaining=100 (priority reset)

# 3. Reduce size to manage risk (priority preserved at 151)
AMEND|ID=abc123...|QTY=80|RTAG=amd-down-002
# → AMENDED: price=151.0 qty=80 remaining=80

# 4. After a partial fill of 30, reduce remaining exposure
#    (original qty was 80, filled 30, so remaining = 50)
AMEND|ID=abc123...|QTY=60|RTAG=amd-partial-001
# → AMENDED: price=151.0 qty=60 remaining=30
#    Note: new_qty (60) - filled (30) = 30 remaining
```

6.13.7 Amendment Events

Topic	Fired When
`order.amended.{GW}`	Amendment accepted; contains new `price`, `qty`, `remaining_qty`, `priority_reset`
`order.ack.{GW}` (`accepted=false`)	Amendment rejected; contains stable `reject_code` plus human-readable `reason`

`priority_reset=true` means the engine assigned the order a new `arrival_seq` (the order fell to the back of the queue at its price level). It says nothing about the order's `timestamp` field, which is client-supplied and unrelated to queue position — see Priority Rules.

6.13.8 A price amend can trigger an immediate fill

If a price amend makes the order **marketable** — the new price crosses the current best opposite price — the engine does not simply leave it resting at the new price. It pulls the resting copy, re-processes it through the normal matching path as if it were a fresh aggressive order, and publishes any resulting `order.fill.{GW}`/`trade.executed` events before the order (or its remainder) rests again. From the caller's point of view this looks like an `order.amended.{GW}` event followed immediately by one or more `order.fill.{GW}` events for the same order ID.

```
# Resting buy at 149.00; best ask is currently 150.00 (no cross)
AMEND|ID=<order-id>|PRICE=150.50|RTAG=amd-cross-001
# → AMENDED: price=150.5 ...   (priority reset)
# → FILL: ...                    (the amend crossed the book and filled immediately)
```

Quantity-only amends never trigger this path — only a price change (with or without an accompanying quantity change) can make an order marketable. If the symbol is halted or the session phase is non-continuous, the amend still applies to the resting order but cannot cross, even if the new price would otherwise be marketable.

6.13.9 Edge Cases and Gotchas

Scenario	Behaviour
Order not found (wrong ID)	**REJECTED** — "Order not found"
Order owned by another gateway	**REJECTED** — "Cannot amend an order owned by another gateway"
Order already filled	**REJECTED** — "Cannot amend FILLED order"
Order already cancelled	**REJECTED** — "Cannot amend CANCELLED order"
MARKET / STOP / FOK order	**REJECTED** — e.g. `"Cannot amend STOP orders"` (message names the actual order type)
Session state is `CLOSED`	**REJECTED** — "Market is closed"
New price violates the symbol's collar	**REJECTED** — same collar-rejection reason a `NEW` order at that price would get
`QTY=` is not a valid integer	**REJECTED** — "Amend quantity must be an integer"
New QTY $\leq$ already filled qty	**REJECTED** — "New quantity must exceed already-filled quantity"
New QTY = 0 or negative	**REJECTED** — "Quantity must be positive"
New PRICE ≤ 0	**REJECTED** — "Price must be positive"
Neither PRICE nor QTY supplied	**REJECTED** — "Amend requires at least PRICE or QTY"
Iceberg order amended	Supported — a price change or qty increase refills the peak to `visible_qty` (priority lost); a qty decrease never enlarges the current peak, only shrinks it to the new remaining
Combo child order amended	Allowed — the child is a normal resting order
OCO leg amended	Allowed — does not affect the OCO linkage
Amend during auction phase	Allowed (any non-`CLOSED` state accepts amends) — see note above about crossing amends

QTY means total quantity

`QTY=` in an `AMEND` command sets the **new total order size**, not the additional quantity you want. For example, if 40 of 100 shares have already filled and you want 80 total, send `QTY=80` (not `QTY=40`).

6.14 OCO (One-Cancels-Other)

Back to Contents

6.14.1 Background

One-Cancels-Other (OCO) is a paired order construct in which two orders are linked such that when one reaches a terminal state (filled, cancelled, or rejected), the other is automatically cancelled. OCO is a core

primitive in institutional trading and has been standardised by the FIX Protocol (`ContingencyType` 1385 = One Cancels the Other) and supported by virtually every major exchange and brokerage since the early 2000s.

The classic use case is a **bracket order**: a trader enters a long position and simultaneously places a take-profit limit above the entry price and a stop-loss below. Once one of them fills, the other is automatically cancelled, preventing the trader from accidentally doubling their position.

6.14.2 Motivation

Use OCO when:

- You want to protect a position with *either* a profit target *or* a stop-loss — whichever happens first — without needing to manually cancel the other leg.
- You want to implement a "bracket" strategy: entry fills, then simultaneously a take-profit limit and a stop-loss stop are live.
- You need the guarantee that you cannot be filled on *both* legs.

OCO vs two independent orders: Without OCO, if your take-profit fills first and you forget to cancel the stop, the stop might trigger later and re-open an unintended position. OCO prevents this by coupling the two orders at the engine level.

OCO vs COMBO: A COMBO (All-Or-Nothing) executes multiple legs simultaneously on a single triggering event. An OCO has two legs that compete — only one can survive.

Educational note: OCO is a powerful risk management tool that automates the lifecycle coupling of two related orders. It is essential for any strategy that relies on mutually exclusive outcomes, such as profit-taking vs stop-loss scenarios.

6.14.3 How traders use it

- **Exit bracket (take-profit / stop-loss).** The classic: one LIMIT to bank the gain, one STOP (or STOP_LIMIT) to cap the loss. See Example 1.
- **Breakout straddle.** A buy stop above a range and a sell stop below it enters whichever way the market leaves the range, and only one side. See Example 5.
- **Two-sided entry.** A buy limit below the market and a sell limit above it work as "buy the dip *or* sell the rip" — one position at most, never both.
- **Exit with a ratchet.** A LIMIT target plus a TRAILING_STOP lets a winner run while keeping a ceiling and a floor. See Example 6.
- **Choose the stop type for the market.** A STOP leg guarantees the exit but may slip; a STOP_LIMIT leg bounds the price but, if the market gaps past its limit, rests unfilled as a limit order. A triggered stop that is merely *resting* has not ended, so its sibling is **not** cancelled: your take-profit LIMIT can still be live next to it. Decide which failure is worse for the position, and cancel the pair yourself if that happens.
- **Partial fills.** A partially filled leg does *not* cancel its sibling; only a leg reaching a terminal state does. After a partial fill you can be working the remaining quantity on one leg and the full quantity on the other; cancel the pair (`CANCEL|OCO_ID=`) if you do not want that.

6.14.4 How It Works

1. The gateway sends an OCO request with two **leg definitions** (side, type, price/stop) plus a shared quantity and `oco_id`.
2. The engine validates both legs (gateway auth, symbol config, price fields), then creates two independent `Order` objects, each tagged with `oco_group_id`.
3. Both orders are posted to the relevant order book and given `NEW` status.
4. The engine publishes an `oco.ack.{GW}` message containing both `order_id_1` and `order_id_2`.
5. When either order transitions to a terminal state (FILLED, CANCELLED, REJECTED), the engine calls `_check_oco_after_event()`:
   - Looks up the sibling order ID.
   - Sends a cancel request to the book for the sibling.
   - Publishes `oco.cancelled.{GW}` with `cancelled_order_id` (the sibling) and `reason` explaining why.
   - Removes the OCO group from the engine's tracking dict.
6. The OCO group can also be cancelled explicitly via `CANCEL|OCO_ID=<id>`, which cancels both legs immediately.

6.14.5 Examples

Example 1 — Classic take-profit + stop-loss bracket (SELL side)

You are long 100 AAPL at $150. You want to: - Sell at $160 (take profit) if the stock rallies, **or** - Sell at $140 (stop-loss) if the stock drops.

```
NEW|TYPE=OCO|OCO_ID=BRACKET1|SYM=AAPL|QTY=100|TIF=GTC|
    LEG1_SIDE=SELL|LEG1_TYPE=LIMIT|LEG1_PRICE=160|
    LEG2_SIDE=SELL|LEG2_TYPE=STOP|LEG2_STOP=140
```

Scenario A: Stock rallies to $162. A buyer crosses the LIMIT leg at $160.

```
Order book receives BUY at 162
LIMIT SELL at 160 → FILLED
Engine: OCO group BRACKET1 — sibling stop cancelled
oco.cancelled.TRADER01 published (cancelled_order_id = stop order)
```

Scenario B: Stock drops to $138. A seller triggers the STOP.

```
Trade at 138 → STOP SELL trigger → converts to MARKET SELL → FILLED
Engine: OCO group BRACKET1 — sibling limit cancelled
oco.cancelled.TRADER01 published (cancelled_order_id = limit order)
```

Example 2 — Two limit orders (buy-side bracket)

You are short 100 AAPL at $150. Cover at $140 (profit) or at $160 (stop).

```
NEW|TYPE=OCO|OCO_ID=SHORT_BRACKET|SYM=AAPL|QTY=100|TIF=GTC|
    LEG1_SIDE=BUY|LEG1_TYPE=LIMIT|LEG1_PRICE=140|
    LEG2_SIDE=BUY|LEG2_TYPE=LIMIT|LEG2_PRICE=160
```

Whichever price is touched first fills; the other is auto-cancelled.

Example 3 — Explicit cancellation of the entire OCO pair

Before either leg fills:

```
CANCEL|OCO_ID=BRACKET1
```

Both legs are cancelled atomically. The engine publishes `order.cancelled` for each leg and removes the OCO group.

Example 4 — A malformed OCO is refused whole

A leg with a missing or unsupported field never reaches the book. Here leg 1 is a `LIMIT` with no price:

```
NEW|TYPE=OCO|OCO_ID=BAD1|SYM=AAPL|QTY=50|TIF=GTC|LEG1_SIDE=SELL|LEG1_TYPE=LIMIT|LEG2_SIDE=SELL|LEG2_TYPE=STOP|LEG
```

The gateway answers with a `MISSING_FIELD` error (and a malformed number such as `LEG1_PRICE=INVALID` is refused the same way, as `INVALID_VALUE`). Neither leg is created. If the pair reaches the engine and fails there, you receive `oco.ack` with `accepted=false` and a `reason`, again with neither leg posted.

Example 5 — Breakout bracket (range trading)

The stock is coiled between 148 and 152 and you have no view on direction, only that it will leave the range. You want to be in on whichever side it breaks, but not both:

```
NEW|TYPE=OCO|OCO_ID=BREAKOUT|SYM=AAPL|QTY=100|TIF=DAY|LEG1_SIDE=BUY|LEG1_TYPE=STOP|LEG1_STOP=152.00|LEG2_SIDE=SELL
```

If a trade prints at 152.00 or higher the buy stop fires, you are long, and the sell stop is cancelled. If the stock instead breaks down through 148.00 the sell stop fires and the buy is cancelled. A stop that is *already* breached when the pair is placed fires on entry and cancels its sibling at once — place the legs on the correct side of the last trade price.

Example 6 — Trailing protection with a profit target

You are long 100 AAPL. Take profit at 160, otherwise let a trailing stop follow the price up:

```
NEW|TYPE=OCO|OCO_ID=RIDE|SYM=AAPL|QTY=100|TIF=GTC|LEG1_SIDE=SELL|LEG1_TYPE=LIMIT|LEG1_PRICE=160.00|LEG2_SIDE=SELL
```

The limit takes the profit at 160 and cancels the trailing stop. If the stock reverses first, the trailing stop sells and the limit is cancelled.

6.14.6 Gateway Syntax

```
# Full OCO request syntax
NEW|TYPE=OCO|OCO_ID=<label>|SYM=<symbol>|QTY=<qty>|TIF=<tif>
   |LEG1_SIDE=<BUY|SELL>|LEG1_TYPE=<order_type>|LEG1_PRICE=<price>
   |LEG2_SIDE=<BUY|SELL>|LEG2_TYPE=<order_type>|LEG2_STOP=<stop_price>

# LIMIT + STOP (take-profit + stop-loss)
NEW|TYPE=OCO|OCO_ID=TPS|SYM=AAPL|QTY=50|TIF=GTC|LEG1_SIDE=SELL|LEG1_TYPE=LIMIT|LEG1_PRICE=155|LEG2_SIDE=SELL|LEG2_

# Two LIMITs (buy bracket)
NEW|TYPE=OCO|OCO_ID=BB|SYM=AAPL|QTY=50|TIF=GTC|LEG1_SIDE=BUY|LEG1_TYPE=LIMIT|LEG1_PRICE=140|LEG2_SIDE=BUY|LEG2_TY

# LIMIT + TRAILING_STOP (take-profit with adaptive stop)
NEW|TYPE=OCO|OCO_ID=TS_BRACKET|SYM=AAPL|QTY=50|TIF=GTC|LEG1_SIDE=SELL|LEG1_TYPE=LIMIT|LEG1_PRICE=160|LEG2_SIDE=SE

# Cancel the entire OCO pair
CANCEL|OCO_ID=TPS
```

TRAILING_STOP legs

`TRAILING_STOP` is a valid OCO leg type. Use `LEG<n>_TRAIL=` for the trail offset and optionally `LEG<n>_STOP=` for the initial stop price (computed from `last_trade_price ± trail_offset` if omitted). `LEG<n>_TRAIL=` is required for any `TRAILING_STOP` leg or the OCO will be rejected.

6.14.7 OCO Events

Topic	Fired When
`oco.ack.{GW}`	OCO pair accepted; contains `order_id_1`, `order_id_2`
`oco.ack.{GW}` (`accepted=false`)	OCO pair rejected; contains `reason`
`oco.cancelled.{GW}`	Sibling auto-cancelled after the other leg settled
`order.fill.{GW}`	Each leg is an independent order; fill events are per-leg
`order.cancelled.{GW}`	Published for each leg on explicit CANCEL|OCO_ID=

6.14.8 Edge Cases and Gotchas

OCO_ID must be unique among your live pairs

Submitting an `OCO_ID` that one of your own pairs still holds is refused (`oco.ack` with `accepted=false`, `OCO id … is already in use`). The label is free again once that pair has ended. Ids are scoped per gateway, so two gateways may use the same label.

Which leg types are allowed. Each leg is `LIMIT`, `STOP`, `STOP_LIMIT`, `TRAILING_STOP`, `MARKET`, `IOC` or `FOK` and takes the same fields as the single order (`LEG<n>_PRICE`, `LEG<n>_STOP`, `LEG<n>_TRAIL`). `ICEBERG` is not allowed (an OCO leg has no `VISIBLE`). Both legs are on the same symbol and share the quantity and `TIF`. An OCO takes no `SMP=`; both legs use the gateway's configured default. Only a resting or dormant leg can usefully wait for its sibling — a `MARKET`, `IOC` or `FOK` leg trades or dies on entry and so ends the pair at once.

Scenario	Behaviour
OCO_ID already in use	**REJECTED** with `oco.ack accepted=false` — neither leg is created
Unknown symbol or gateway	**REJECTED** with `oco.ack accepted=false`
LIMIT leg missing `LEG1_PRICE=` or `LEG2_PRICE=`	**REJECTED**
STOP leg missing `LEG1_STOP=` or `LEG2_STOP=`	**REJECTED**
Partial fill of one leg	Does NOT cancel the sibling — only a terminal state (FILLED, CANCELLED, REJECTED) does
Both legs fill simultaneously	First fill to be processed cancels the second; race resolved by event order

A leg ends while an order is matching	Filled, cancelled by self-match prevention, an IOC leg's unfilled rest, or a stop leg short of liquidity: the sibling is pulled at once, before any stop that order triggers can trade with it — even a stop leg already triggered and waiting its turn. So `oco.cancelled` can arrive *before* the leg's own `order.fill` or `order.cancelled`
A `FOK` meets both legs of a pair	Only the first leg it meets counts as available: once that leg fills, the other is gone
Leg 1 fills, is cancelled or rejected on entry	Leg 2 is acknowledged, then cancelled with `oco.cancelled`; it never reaches the book
Symbol halted	Legs rest without matching, as ordinary limit orders do. A `MARKET`, `IOC` or `FOK` leg cannot rest and is rejected with `INSTRUMENT_HALTED` (or `CIRCUIT_BREAKER_ACTIVE`); outside continuous trading, with `SESSION_NOT_PERMITTED`
GTC OCO across sessions	Both legs persist with their TIF; if GTC they survive session reset

6.15 See also

In this book:

- Combos — multi-symbol multi-leg strategies with cascade cancellation
- ALF Console — the full command syntax for submitting these order types

Operator's Guide:

- Auctions & Scheduling — which order types are accepted in each session phase
- Persistence — how GTC orders survive engine restarts
- Risk Controls — how collars and halts affect order admission

Protocols and Clients:

- ALF Protocol Reference — formal grammar for all order fields

CHAPTER 7

Combo Orders

Learning objectives

After reading this page you will understand:

- Why single-leg orders are not enough for many real trading strategies, and what **leg risk** means in practice
- How combo orders solve that problem by linking multiple child orders as a single atomic unit
- The real-world strategies that require combos — from simple pairs trades to three-leg statistical arbitrage — explained without jargon
- The extra complexity combos introduce: distributed state, cascade cancellation, fill-event hooks, and GTC persistence recovery

7.1 What Is a Combo Order, and Why Would I Use One?

Imagine you are watching two tech stocks, MSFT and AAPL, and you have noticed that whenever MSFT rises faster than AAPL the gap tends to close within a few days. You want to bet on that gap closing: buy MSFT, sell AAPL, at the same moment. If you type two separate orders — one buy, one sell — the market can move between the two keystrokes. The MSFT buy fills, AAPL doesn't, and you are stuck owning something you didn't want to own alone.

That is called **leg risk**, and it is why combo orders exist.

A combo order bundles two or more orders into one instruction that the exchange treats as a unit:

- All legs are submitted together.
- If the market cannot fill all of them, none fill.
- If any leg is cancelled, rejected or expires, the exchange automatically cancels all the remaining legs for you.

7.1.1 Three real-world situations where combos matter

Situation 1 — You want to trade a relationship, not a direction. You do not care whether the stock market goes up or down today. You care that MSFT is temporarily cheap *relative to* AAPL. A combo lets you express that view in one trade: buy MSFT, sell AAPL. You profit when the gap narrows, regardless of

which way the whole market moves.

Situation 2 — You want to enter a position and hedge it at the same time. You are bullish on AAPL but worried about a broad market crash. A combo lets you buy AAPL and simultaneously sell an index (like SPY) in one atomic step. If the whole market falls, your SPY short offsets the AAPL loss.

Situation 3 — You are a professional quant with a model. Your algorithm says a particular three-stock basket is mispriced by 0.3%. You need all three legs to fill at the same time, otherwise the arbitrage disappears before you can complete the position. A three-leg combo is the only practical way to do this.

Until now every order in EduMatcher has been a **single-leg** order — one symbol, one side, one quantity. That is enough to teach matching, price-time priority, and basic order types. Real markets, however, are full of strategies that require **multiple orders to work together as a unit**. That is what combo orders are for.

7.2 What Are Combo Orders?

A combo order (also called a multi-leg order, spread order, or strategy order) is a single instruction that bundles **two or more child orders** across different symbols. The child orders are linked: they are submitted, tracked, and — critically — cancelled as a group.

```
COMBO [BUY 100 MSFT @415.00, SELL 100 AAPL @210.00]
```

The exchange decomposes this into two child limit orders and posts them to the respective order books. The parent combo tracks whether all legs have filled.

Real-world terminology

CME calls these **spread orders**. NASDAQ uses **complex orders**. Options exchanges speak of **strategy orders** (straddles, strangles, butterflies). The mechanics are the same: multiple legs, one intent.

7.3 Why Do Combo Orders Exist?

7.3.1 The problem with placing legs separately

Suppose you want to buy MSFT and simultaneously sell AAPL — a classic **pairs trade**. With single-leg orders you would type two commands:

```
NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=415.00
NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=210.00
```

Between the first and second keystroke, the market can move. Your MSFT buy might fill ("fill" = the exchange matched your order against a counterparty and the trade executed) at 415.00 but by the time your AAPL sell arrives, the price has dropped to 208.00 and your limit sits unfilled. You are now **unhedged** — **long** MSFT (you bought it and own it) with no offsetting **short** (a position where you profit if the price falls). Hedging means holding a second position that offsets the risk of your first; without the AAPL short, you have pure directional MSFT exposure. This is called **leg risk** (or execution risk).

7.3.2 What combos solve

Single-leg problem	Combo solution
Leg risk — one side fills, the other doesn't	Both legs submitted atomically in one message
Manual cancellation — if one leg fails you must remember to cancel the other	Cascade-cancel: if any leg is cancelled, rejected or expires, all siblings are cancelled automatically
State tracking — you mentally track which orders belong together	The engine tracks the parent combo and its children as a unit
Strategy intent — the engine doesn't know your two orders are related	The engine understands the relationship and can report combo-level status

7.4 New Strategies Enabled by Combos

Single-leg trading limits you to directional bets: buy because you think the price will rise, sell because you think it will fall. Combos unlock **relative-value** and **volatility** strategies where the profit comes from the relationship between two or more instruments, not the direction of any one.

7.4.1 Pairs Trading

Buy one stock, sell a correlated stock. You profit when the spread between them reverts to its historical mean, regardless of whether the overall market moves up or down.

```
COMBO: BUY 100 MSFT @415.00, SELL 100 AAPL @210.00
```

Thesis: MSFT is temporarily cheap relative to AAPL. If the gap narrows — whether both rise, both fall, or MSFT rises faster — the combo profits.

7.4.2 Calendar (Time) Spread

Buy a near-dated instrument and sell a far-dated one (or vice versa). Common in futures and options. In EduMatcher this maps to two symbols representing different expiries.

```
COMBO: BUY 50 ESM26 @5200.00, SELL 50 ESU26 @5220.00
```

Thesis: The price difference between June and September futures will widen or narrow predictably as expiry approaches.

7.4.3 Statistical Arbitrage

Trade a basket of instruments whose combined position has near-zero market exposure. Requires more legs than pairs trading.

```
COMBO: BUY 200 AAPL @210.00, SELL 100 MSFT @415.00, SELL 50 GOOG @170.00
```

Thesis: A quantitative model says this basket is mispriced. The combo lets you enter the entire position atomically.

7.4.4 Hedged Entry

Buy a stock and simultaneously place a protective stop on a correlated instrument to limit downside.

```
COMBO: BUY 100 MSFT @415.00, SELL 100 SPY @520.00
```

Thesis: You want MSFT exposure but hedge broad market risk by shorting the index.

7.4.5 Market-Making Spread

A **market maker** is a participant whose job is to continuously post both a buy order (bid) and a sell order (ask) on the book, providing liquidity so that other traders can execute immediately. The market maker profits from the small price gap between their bid and ask (the "bid-ask spread"), but takes on inventory risk.

With combos, a market maker can quote both sides of two related symbols simultaneously, capturing the bid-ask spread on both while maintaining a hedged book.

```
COMBO: BUY 100 MSFT @414.80, SELL 100 MSFT_PROXY @415.20
```

7.5 Complexity Introduced by Combos

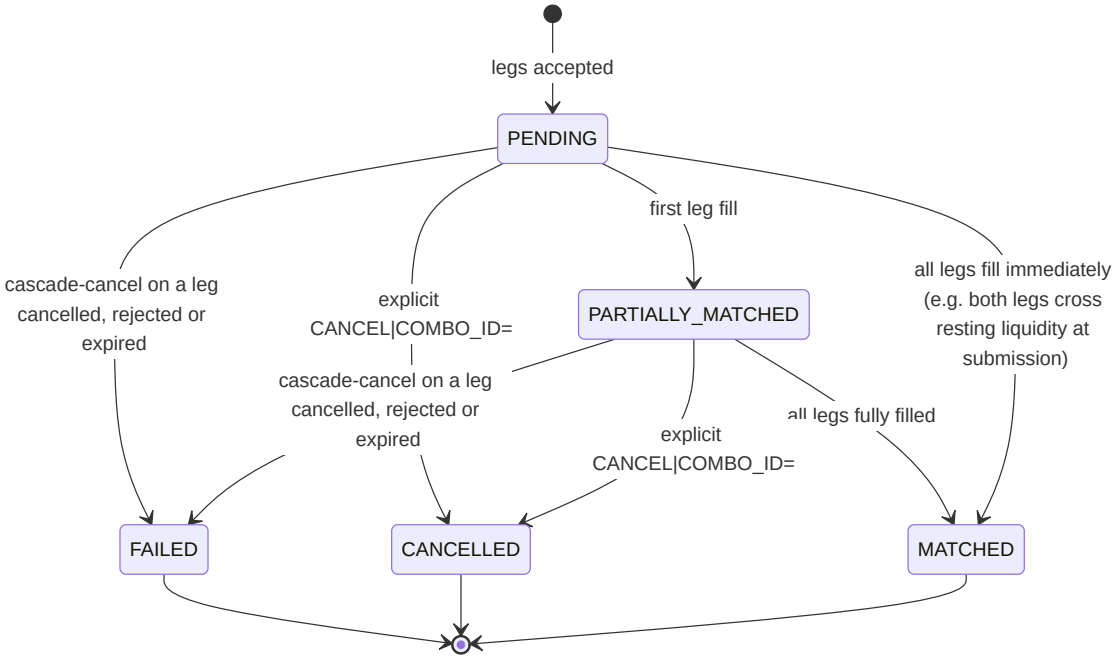
Combos look simple on the surface — "just submit two orders together." Under the hood they introduce **significant new complexity** that single-leg orders never had to deal with.

7.5.1 Distributed State

A single-leg order lives in one order book. A combo spans **multiple books**. The engine must track a parent object (the combo) and its children (the legs) across separate data structures. Every fill, cancel, or expiry on any child must propagate back to the parent.

7.5.2 Lifecycle Management

Single-leg orders have a simple lifecycle: NEW → PARTIAL → FILLED (or CANCELLED / EXPIRED). Combos add a **second layer**:



The engine must detect transitions at the combo level whenever a child-level event occurs.

7.5.3 Cascade Cancellation

If one leg of a combo is cancelled, rejected or expires, the remaining legs must be **automatically cancelled**. This is non-trivial:

- The cancelled leg might be on a different book than its siblings.
- A sibling might be **partially filled** — do you cancel only the unfilled remainder, or reject the entire combo?
- A sibling might fill in the same engine cycle that the first leg is cancelled — a race between fill and cancel.

EduMatcher handles this by marking the combo as FAILED and cancelling all siblings' unfilled quantities. Fills that already occurred are **not** unwound.

No trade unwinds

Once a trade is executed, it is final. A combo failure does not reverse fills that already happened on other legs. This mirrors real exchange behavior — trades are irrevocable.

7.5.4 Fill Event Hooks

In a single-leg system the engine processes an order and publishes fills. With combos, every fill on a resting child order must also trigger a **parent combo check**: "Is this combo now fully matched?" This requires a reverse lookup from child order ID to parent combo ID on every fill event.

7.5.5 Persistence Recovery

GTC combos must survive engine restarts. On shutdown the engine persists both the parent combo metadata and its child orders. On startup it must reconstruct the parent-child links and resume tracking fill progress — a non-trivial deserialization problem.

7.5.6 Visibility and Information Leakage

Should combo child orders be visible on the per-symbol book? If yes, counterparties can see "someone is building a pairs position" and **front-run** the trade (rush to trade ahead of them, anticipating the price impact of the remaining legs). If no, the book underrepresents true liquidity. EduMatcher shows child orders on the book (they are normal resting orders) but does **not** label them as combo legs.

7.6 Trading Combos in EduMatcher

7.6.1 Submitting a Combo

From the gateway terminal, use the `TYPE=COMBO` format:

```
NEW|TYPE=COMBO|COMBO_ID=PAIR-001|COMBO_TYPE=AON|TIF=GTC|LEG_COUNT=2|LEG0.SYM=MSFT|LEG0.SIDE=BUY|LEG0.QTY=100|LEG0
```

Field	Meaning
`TYPE=COMBO`	Tells the parser this is a multi-leg order
`COMBO_ID=PAIR-001`	Your tracking label (human-readable)
`COMBO_TYPE=AON`	All-or-none semantics: combo completes only when **all** legs are fully filled
`TIF=GTC`	Time-in-force applied to all legs
`LEG_COUNT=2`	Number of legs (2–10)
`LEG<i>.SYM`	Symbol for leg i
`LEG<i>.SIDE`	BUY or SELL for leg i
`LEG<i>.TYPE`	Order type for leg i — `LIMIT`, `MARKET`, `STOP`, `STOP_LIMIT`, `FOK`, or `IOC` (defaults to `LIMIT` if omitted). `ICEBERG` and `TRAILING_STOP` are not supported on a combo leg
`LEG<i>.QTY`	Quantity for leg i
`LEG<i>.PRICE`	Limit price for leg i — required for legs typed `LIMIT`, `IOC`, `FOK` or `STOP_LIMIT`; refused on `MARKET` and `STOP`
`LEG<i>.STOP`	Stop price for leg i — required for legs typed `STOP` or `STOP_LIMIT`; refused on other types
`SMP=`	Optional self-match-prevention action — `NONE`, `CANCEL_AGGRESSOR`, `CANCEL_RESTING`, or `CANCEL_BOTH`. One value for the *whole combo*, applied identically to every leg (there is no per-leg `LEG<i>.SMP` on this protocol). If omitted, falls back to the gateway's configured `participants[].smp_action` (else `NONE`) — see Risk Controls — Self-Match Prevention for the full mechanics and precedence rules

One field rule for every entry point

The price/stop requirements above are the same for a live combo from any gateway and for a combo seeded from `market_maker_combos` in `engine_config.yaml`: the engine applies the same per-type field table (Order Types — Fields each type takes) and rejects the whole combo, with a descriptive error, rather than silently accept an inconsistent leg.

7.6.2 What Happens After Submission

1. The engine validates the combo (symbols allowed, no duplicates, valid quantities).
2. If valid, the engine sends a **combo ACK** back to your gateway.
3. The engine creates **child orders** — one per leg, in leg order — and posts them to the respective order books. Every leg event and `combo.status` therefore follows the `combo.ack`.
4. Child orders match and rest like normal orders. Each fill is reported to you as a standard `order.fill` event.
5. When **all legs are fully filled**, the engine publishes a `combo.status(MATCHED)` event.
6. If any leg is **cancelled, rejected or expires** before all legs fill, the engine cascade-cancels the remaining legs and publishes `combo.status(FAILED)`. This holds on entry too: a leg rejected or cancelled as it is posted fails the combo at once, the legs already posted are cancelled, and the legs after it are never posted.

A `MARKET`, `IOC` or `FOK` leg cannot rest, so while its symbol is not matching it is rejected, with the code an ordinary order of that type would get: `INSTRUMENT_HALTED` (or `CIRCUIT_BREAKER_ACTIVE`) on a halted symbol, `SESSION_NOT_PERMITTED` outside continuous trading. An all-or-none combo that cannot fill every leg in full is held back: no leg matches, and the legs rest as interest. If a leg would trade at once, the combo is rejected instead: its first leg gets `INSUFFICIENT_LIQUIDITY` and no other leg is posted, since a held-back marketable leg would rest crossed with the book. A `MARKET`, `IOC` or `FOK` leg of a held-back combo is rejected with `INSUFFICIENT_LIQUIDITY` too, as is a `FOK` leg short of liquidity. Each way the combo fails (step 6).

7.6.3 Cancelling a Combo

```
CANCEL|COMBO_ID=PAIR-001
```

This cancels all child legs atomically. Fills that already occurred are not reversed.

7.6.4 Monitoring Combo Status

Child orders appear in the `ORDERS` command's output like any other resting order (the `ORDERS` response does not currently include a parent-combo-ID column). To follow a combo's overall progress, watch for `combo.ack` and `combo.status` events on your gateway — see Messages.

7.7 New Demands on Market Makers

Single-leg market makers quote bid and ask on one symbol. Combos create demand for **cross-symbol liquidity** — someone must be willing to take the other side of both legs simultaneously.

7.7.1 Why Single-Leg MM Is Not Enough

A pairs trader submits `COMBO [BUY MSFT, SELL AAPL]`. The MSFT leg rests on the MSFT book and the AAPL leg rests on the AAPL book. A single-symbol market maker on MSFT might fill the MSFT leg, but nobody

fills the AAPL leg. The combo sits partially matched, exposed to leg risk, until eventually the AAPL leg fills or expires.

7.7.2 What Combo-Aware Market Makers Provide

A market maker that understands combos can:

- **Seed both sides** of common pairs at startup, ensuring combos find immediate liquidity.
- **Quote spreads** rather than individual symbols, tightening the effective cost of entering a combo position.
- **Rebalance dynamically** — when one leg fills, adjust quotes on the other leg to maintain a hedged book.

7.7.3 Configuration

In `engine_config.yaml`, market-maker startup quote seeds can be combined with combo seeds:

```
symbols:
  MSFT:
    market_maker_quotes:
      - gateway_id: MM01
        bid_price: 414.00
        ask_price: 416.00
        bid_qty: 100
        ask_qty: 100
  AAPL:
    market_maker_quotes:
      - gateway_id: MM01
        bid_price: 209.00
        ask_price: 211.00
        bid_qty: 100
        ask_qty: 100

market_maker_combos:
  - combo_id: MM-MSFT-AAPL
    combo_type: AON
    tif: GTC
    legs:
      - symbol: MSFT
        side: BUY
        order_type: LIMIT
        quantity: 100
        price: 414.50
      - symbol: AAPL
        side: SELL
        order_type: LIMIT
        quantity: 100
        price: 210.50
```

This ensures that when a retail trader submits a combo, there is resting liquidity on both legs from day one.

Note that a `market_maker_combos` entry has no `gateway_id` field — unlike `market_maker_quotes`, it is not attached to `MM01` or any other configured gateway. The engine assigns every config-seeded combo the fixed

internal owner `MM`, which is not a real, authenticatable gateway ID. As a result, a seeded combo cannot be cancelled with `CANCEL|COMBO_ID=` from any ALF console session — cancellation only works for combos submitted live by an authenticated gateway.

7.8 Example Strategies in Detail

7.8.1 Example 1 — Pairs Trade: MSFT vs AAPL

Setup: You believe MSFT is undervalued relative to AAPL. The historical price ratio is 2.0 (MSFT 2× AAPL), but today MSFT is trading at 415 and AAPL at 212 — a ratio of 1.96. You expect reversion.

Combo:

```
NEW|TYPE=COMBO|COMBO_ID=PAIR-REVERT|COMBO_TYPE=AON|TIF=GTC|LEG_COUNT=2|LEG0.SYM=MSFT|LEG0.SIDE=BUY|LEG0.QTY=100|LE
```

Outcome if ratio reverts to 2.0:

P&L convention: for your **long** MSFT position, profit = (exit price − entry price) × quantity. For your **short** AAPL position, profit = (entry price − exit price) × quantity (you profit when the price falls below where you sold).

Scenario	MSFT	AAPL	P&L
Both rise	420 (+5)	210 (+2)	+500 + 200 = **+700**
Both fall	410 (−5)	205 (+7)	−500 + 700 = **+200**
MSFT rises, AAPL flat	420 (+5)	212 (0)	+500 + 0 = **+500**

The combo profits regardless of market direction — only the spread matters.

7.8.2 Example 2 — Hedged Long

Setup: You are bullish on AAPL but worried about a broad market sell-off.

Combo:

```
NEW|TYPE=COMBO|COMBO_ID=HEDGE-LONG|COMBO_TYPE=AON|TIF=DAY|LEG_COUNT=2|LEG0.SYM=AAPL|LEG0.SIDE=BUY|LEG0.QTY=200|LE
```

If the market crashes, your SPY short offsets losses on AAPL. If AAPL outperforms the index, you profit on the spread.

7.8.3 Example 3 — Calendar Spread

Setup: You expect the June/September ES futures spread to narrow.

Combo:

```
NEW|TYPE=COMBO|COMBO_ID=CAL-ES|COMBO_TYPE=AON|TIF=GTC|LEG_COUNT=2|LEG0.SYM=ESM26|LEG0.SIDE=BUY|LEG0.QTY=50|LEG0.PR
```

You enter at a spread of 20 points. If the spread narrows to 10, you close both legs and pocket 10 × 50 = 500 points profit.

7.8.4 Example 4 — Three-Leg Statistical Arbitrage

Setup: A quant model identifies a mispricing in the AAPL/MSFT/GOOG triangle.

Combo:

```
NEW|TYPE=COMBO|COMBO_ID=STAT-ARB|COMBO_TYPE=AON|TIF=GTC|LEG_COUNT=3|LEG0.SYM=AAPL|LEG0.SIDE=BUY|LEG0.QTY=200|LEG0
```

The combo ensures all three legs enter simultaneously. Without a combo, the arbitrageur risks partial execution and model-breaking slippage.

7.8.5 Example 5 — Market-Maker Spread Seeding

Setup: A market maker wants to provide liquidity for MSFT/AAPL pairs traders.

Combo (bid side of spread):

```
NEW|TYPE=COMBO|COMBO_ID=MM-BID|COMBO_TYPE=AON|TIF=GTC|LEG_COUNT=2|LEG0.SYM=MSFT|LEG0.SIDE=BUY|LEG0.QTY=100|LEG0.PR
```

Combo (ask side of spread):

```
NEW|TYPE=COMBO|COMBO_ID=MM-ASK|COMBO_TYPE=AON|TIF=GTC|LEG_COUNT=2|LEG0.SYM=MSFT|LEG0.SIDE=SELL|LEG0.QTY=100|LEG0.
```

The market maker earns the spread between bid and ask on **both** legs — but must manage the risk that one leg fills before the other.

7.9 Summary

Aspect	Single-Leg Orders	Combo Orders
Scope	One symbol	2–10 symbols
Leg risk	N/A	Managed by cascade-cancel
Strategies	Directional (long/short)	Relative-value, hedged, arbitrage
State tracking	One book, one lifecycle	Parent combo + children across books
Cancellation	Cancel one order	Cancel cascades to all siblings
Market-maker demands	Quote one symbol	Quote cross-symbol spreads
Persistence	One order per file entry	Combo metadata + child orders
Complexity	Low	Significantly higher

Combos are where an educational trading system transitions from "toy" to "architecturally realistic." They force you to think about distributed state, failure cascades, and the tension between atomicity and performance — the same problems real exchanges spend years solving.

7.10 See also

In this book:

- Implied Orders - A crash course in implied (or synthetic) orders
- Order Types — the individual order types used as combo legs

- ALF Console — the full COMBO and CANCEL|COMBO_ID= command syntax

Operator's Guide:

- Persistence — how GTC combos are saved and restored across sessions

Protocols and Clients:

- Messages — `combo.ack`, `combo.status`, `order.fill`, and cascade-cancel events

CHAPTER 8

Implied Orders (a Concept Not Yet Implemented)

Learning objectives

After reading this page you will understand:

- What an **implied (synthetic) order** is, and why exchanges with related instruments (spreads, baskets) generate them automatically from existing outright liquidity
- How to derive an implied bid and ask from the top-of-book of the instruments it is built from, for a simple 1:1 spread and for a general weighted multi-leg synthetic
- How implied quantity is limited by the smallest leg after ratio normalization
- That this is a **conceptual, not-yet-implemented** mechanism in EduMatcher today — what exists instead is parent/child combo tracking, covered in Combo Orders

This section explains an advanced but very important exchange concept: **implied orders** (also called **synthetic orders**).

Not supported

This concept is not yet supported by EduMatcher Current EduMatcher combo handling is parent/child leg tracking over outright books. It does not yet implement a native synthetic instrument book with implied quote generation. This section describes the exchange-grade mechanism typically added in a later architectural step.

8.1 The key idea in one paragraph

Some exchanges maintain a **spread book** — a separate order book whose "instrument" is not a single stock but a *price difference* between two (or more) stocks. For example, a spread book for "MSFT minus AAPL" lets traders buy or sell the numerical difference between MSFT and AAPL prices as if it were its own tradeable product. Concretely, "buying the spread at 205" means you end up long MSFT and short AAPL at a net cost of 205 per share — regardless of whether MSFT is at 415 and AAPL at 210, or MSFT at 420 and AAPL at 215. You are trading the *gap*, not the individual prices.

Now imagine that spread book is empty — nobody has directly placed an order in it. Yet on the **outright** MSFT book (the ordinary, single-symbol order book where people trade just MSFT) there is a resting bid at 415.00 (meaning someone is willing to buy MSFT at that price), and on the outright AAPL book there is a

resting ask at 210.10 (someone is willing to sell AAPL at that price).

Here is the key insight: any trader who wants to **sell** the spread (= sell MSFT and buy AAPL) could execute against those two existing orders right now — sell MSFT at 415.00 (hitting that bid) and buy AAPL at 210.10 (hitting that ask) — netting $415.00 - 210.10 = 204.90$. The exchange recognizes this and automatically publishes a **synthetic bid of 204.90** in the spread book. That synthetic quote is an **implied order** — executable liquidity that exists but was never typed by any human.

Major derivatives exchanges (CME Globex, Eurex T7, ICE) rely heavily on implied order engines to connect fragmented books and improve fill rates.

If you are new to markets, read this as:

- A **direct order** is something a trader explicitly entered.
- An **implied/synthetic order** is extra liquidity the exchange computes from other available orders.

8.2 First: beginner terminology

- **Liquidity**: how much you can trade without waiting. More resting orders means more liquidity.
- **Bid**: highest price someone is willing to buy at.
- **Ask**: lowest price someone is willing to sell at.
- **Bid-ask spread**: the gap between the best ask and best bid in a single book (for example, if MSFT best bid is 415.00 and best ask is 415.20, the bid-ask spread is 0.20).
- **Spread instrument**: a synthetic product defined as the price difference between two or more symbols (for example, "MSFT minus AAPL"). Not the same as bid-ask spread — this is a tradeable product in its own right.
- **Resting order**: an order that has been posted to the book but not yet matched. It sits there passively, waiting for someone to trade against it.
- **Hitting**: executing against a resting order. "Hit the bid" means you sell to the person whose buy order is resting at the bid price. "Hit the ask" (also called "lifting the offer") means you buy from the person whose sell order is resting at the ask price.
- **Leg**: one component of a combo. A 2-leg combo has two child orders.
- **Outright**: a normal single-symbol market (for example, just MSFT).
- **Synthetic instrument**: a derived instrument built from other instruments (for example, spread = MSFT - AAPL).
- **Implied order**: a quoted buy/sell order in one book that is inferred from prices resting in related books.
- **Atomic execution**: all required legs execute together as one transaction, or none execute.

8.3 Why combo markets are hard without implied orders

Combo markets have structural problems:

1. Fragmented liquidity: Liquidity exists in many places (outright books, other combo books), so one combo book can look empty even when executable prices exist elsewhere.
2. Leg risk: If a strategy is filled leg-by-leg manually, one leg can fill while others do not. The trader is left with unwanted directional exposure.
3. Wide displayed spreads: Without implied quotes, visible combo bid/ask can be sparse and wide, discouraging participation.

4. Price inconsistency: The same economic position can be priced differently across related markets, creating temporary dislocations.

8.4 What problem implied orders solve

Implied orders solve a visibility and matching problem:

- They expose executable combo liquidity that already exists indirectly in related books.
- They tighten quoted spreads in combo books.
- They increase fill probability for strategy traders.
- They align related market prices more quickly.

Important: implied orders do not create "free money". They reveal and route existing executable relationships between books.

8.5 How implied orders accomplish this

At a high level, the exchange continuously does this:

1. Watch all relevant books (outrights and related combos).
2. Build candidate synthetic prices using valid price relationships.
3. Compute the maximum executable synthetic quantity from limiting legs.
4. Publish implied bid/ask levels in the target book.
5. Recompute immediately whenever any source book changes.

When a trader hits an implied quote, the exchange does not simply report a theoretical price — it actually **executes** the underlying leg trades. For example, if a trader sells the MSFT-AAPL spread at the implied bid of 204.90, the exchange simultaneously:

1. Sells MSFT at 415.00 against the resting bid in the MSFT book.
2. Buys AAPL at 210.10 against the resting ask in the AAPL book.
3. Reports all fills to the respective parties.

This happens atomically — either both legs execute, or neither does. The original resting orders in the outright books are consumed (partially or fully), just as if someone had traded against them directly.

8.6 Detailed calculation model (2-leg spread)

Assume a spread instrument:

$$S = A - B$$

where:

- A is leg 1 (for example MSFT)
- B is leg 2 (for example AAPL)
- S is the synthetic spread market

Let:

- A_{bid} = best bid price in A

- A_{ask} = best ask price in A
- B_{bid} = best bid price in B
- B_{ask} = best ask price in B

8.6.1 Implied bid for spread S

The implied bid represents resting synthetic buy interest — the price a spread **seller** can receive by executing against existing outright liquidity.

To sell $S = A - B$, the seller:

- sells A (hits the resting bid → receives A_{bid}), and
- buys B (hits the resting ask → pays B_{ask})

Net price received (= implied bid):

$$S_{bid}^{impl} = A_{bid} - B_{ask}$$

8.6.2 Implied ask for spread S

The implied ask represents resting synthetic sell interest — the price a spread **buyer** must pay to execute against existing outright liquidity.

To buy $S = A - B$, the buyer:

- buys A (hits the resting ask → pays A_{ask}), and
- sells B (hits the resting bid → receives B_{bid})

Net price paid (= implied ask):

$$S_{ask}^{impl} = A_{ask} - B_{bid}$$

8.6.3 Implied quantity

Implied size is constrained by the smallest executable leg quantity after ratio normalization.

For 1:1 ratio spreads:

$$Q_{impl} = \min(Q_A, Q_B)$$

For general ratios $r_A : r_B$:

$$Q_{impl} = \min\left(\left\lfloor \frac{Q_A}{r_A} \right\rfloor, \left\lfloor \frac{Q_B}{r_B} \right\rfloor\right)$$

8.7 Generalized weighted synthetic pricing

For a multi-leg synthetic:

$$P_{synthetic} = \sum_{i=1}^{n} w_i P_i$$

where:

- w_i are signed leg weights (positive for bought legs, negative for sold legs)
- P_i are the price used for leg i, selected according to the side-selection rule below

Side-selection rule: the key question is "which price (bid or ask) do I use for each leg?" The answer depends on *who is executing* and *which direction they trade each leg*:

Computing…	For legs you BUY	For legs you SELL
Synthetic **bid** (what a seller receives)	Use the **ask** (seller must buy this leg, hitting the ask)	Use the **bid** (seller sells this leg, hitting the bid)
Synthetic **ask** (what a buyer pays)	Use the **ask** (buyer buys this leg, hitting the ask)	Use the **bid** (buyer sells this leg, hitting the bid)

Simplified: always use the price that the *taker* would actually receive or pay — the bid when selling into it, the ask when buying into it. This ensures the synthetic price reflects real executable costs, not theoretical mid-prices.

8.8 Worked Example 1 (2-leg, equal ratios)

Goal: build implied quotes for spread $S = MSFT - AAPL$ (1:1).

Assume current books:

- MSFT best bid = 415.00 (qty 120)
- MSFT best ask = 415.20 (qty 80)
- AAPL best bid = 210.00 (qty 90)
- AAPL best ask = 210.10 (qty 200)

Implied spread bid:

$$S_{bid}^{impl} = 415.00 - 210.10 = 204.90$$

Implied spread ask:

$$S_{ask}^{impl} = 415.20 - 210.00 = 205.20$$

Implied bid quantity (1:1, using MSFT bid qty and AAPL ask qty):

$$Q_{bid}^{impl} = \min(120, 200) = 120$$

Implied ask quantity (1:1, using MSFT ask qty and AAPL bid qty):

$$Q_{ask}^{impl} = \min(80, 90) = 80$$

So the spread book can display synthetic liquidity:

- Bid 204.90 x 120
- Ask 205.20 x 80

Even if no trader directly entered spread orders.

8.9 Worked Example 2 (3-leg with ratios)

Synthetic instrument:

$$X = 2 \cdot A - 1 \cdot B - 3 \cdot C$$

Interpretation per 1 unit of X:

- Buy 2 of A
- Sell 1 of B
- Sell 3 of C

Assume top-of-book:

- A bid/ask: 100.00 / 100.20, qty at ask = 250, qty at bid = 180
- B bid/ask: 40.10 / 40.30, qty at bid = 140, qty at ask = 300
- C bid/ask: 10.00 / 10.05, qty at bid = 500, qty at ask = 220

8.9.1 Synthetic bid for X

The implied bid is what a **seller** of X receives. Selling X means reversing the definition: sell A, buy B, buy C.

- sell A at $A_{bid} = 100.00$ (qty available: 180)
- buy B at $B_{ask} = 40.30$ (qty available: 300)
- buy C at $C_{ask} = 10.05$ (qty available: 220)

Price:

$$X_{bid}^{impl} = 2(100.00) - 1(40.30) - 3(10.05)$$

$$= 200.00 - 40.30 - 30.15 = 129.55$$

Quantity limits by ratio:

- A supports $\lfloor 180/2 \rfloor = 90$ synthetic units
- B supports $\lfloor 300/1 \rfloor = 300$ synthetic units
- C supports $\lfloor 220/3 \rfloor = 73$ synthetic units

So:

$$Q_{bid}^{impl} = \min(90, 300, 73) = 73$$

8.9.2 Synthetic ask for X

The implied ask is what a **buyer** of X pays. Buying X means executing the definition directly: buy A, sell B, sell C.

- buy A at $A_{ask} = 100.20$ (qty available: 250)
- sell B at $B_{bid} = 40.10$ (qty available: 140)
- sell C at $C_{bid} = 10.00$ (qty available: 500)

Price:

$$X_{ask}^{impl} = 2(100.20) - 1(40.10) - 3(10.00)$$

$$= 200.40 - 40.10 - 30.00 = 130.30$$

Quantity limits:

- A supports $\lfloor 250/2 \rfloor = 125$
- B supports $\lfloor 140/1 \rfloor = 140$
- C supports $\lfloor 500/3 \rfloor = 166$

So:

$$Q_{ask}^{impl} = \min(125, 140, 166) = 125$$

This yields implied quotes for X:

- Bid 129.55 x 73
- Ask 130.30 x 125

The implied spread is 0.75, representing normal market conditions. With different input prices, implied markets can **cross** (bid ask), indicating an arbitrage that production engines immediately consume via matching.

8.10 Directions of implication

The examples above demonstrate **implied-in**: outright book liquidity is combined to generate synthetic quotes in a combo/spread book. Production exchanges support additional implication directions:

Direction	Source → Target	Example
Implied-in	Outrights → Spread	MSFT bid + AAPL ask → MSFT-AAPL spread bid

Direction	Source → Target	Example
Implied-out	Spread → Outrights	A resting spread order implies an outright quote in one leg, given liquidity in the other
Implied-through	Spread → Spread	Liquidity in spread AB and outright B together imply a quote in spread AC

Implied-out example: A trader posts "sell MSFT-AAPL spread at 205.00." If AAPL has a resting bid at 210.00, the exchange can derive an implied offer in MSFT at 415.00 (= 205.00 + 210.00). This publishes synthetic sell liquidity in the outright MSFT book that did not previously exist.

Implied-through chains multiple relationships: spread AB liquidity combined with outright B liquidity generates implied quotes in spread AC. The combinatorial explosion of paths is the primary reason implied engines are computationally expensive.

Most exchange implied engines support implied-in and implied-out. Only the most sophisticated (CME Globex, Eurex T7) additionally support implied-through, often limited to a configurable hop depth to bound computation.

8.11 Practical implementation challenges

Building implied orders correctly in a matching engine is hard. Common issues:

1. Atomic routing: Implied fills must map to real leg executions in one atomic transaction.
2. Double counting: The same leg liquidity can be reachable through multiple synthetic paths. Engines must prevent over-committing that liquidity.
3. Priority policy: Venues define whether direct orders outrank implied orders at same price/time.
4. Performance: Every book update can trigger recomputation across many related instruments.
5. Consistency under concurrency: Between quote calculation and execution, source books can change. Matching must lock/sequence state safely.
6. Explainability: Traders and surveillance systems need audit trails showing which source liquidity created each implied fill.

8.12 How this relates to EduMatcher today

Current EduMatcher combo handling is parent/child leg tracking over outright books. It does not yet implement a native synthetic instrument book with implied quote generation. This section describes the exchange-grade mechanism typically added in a later architectural step.

CHAPTER 9

Market-Maker Quotes

Learning objectives

After reading this page you will understand:

- What a two-sided market-maker quote is and how it differs from single orders
- How EduMatcher represents one quote as two linked orders on the same symbol
- How quote lifecycle events work: active, inactive-on-fill, cancelled
- How gateway disconnect policy and kill-switch controls interact with quotes
- How to submit, replace, and cancel quotes from the gateway CLI
- Why startup seeding and gateway connectivity are independent events
- What happens when a fill occurs against a seed quote before the MM gateway connects
- How configured circuit breakers can still halt a symbol after fills against seed liquidity

9.1 Concept

A market-maker quote is a coordinated bid/ask pair for one symbol:

- Bid leg: willingness to buy at bid price
- Ask leg: willingness to sell at ask price
- Shared quote ID: lets the engine manage the pair as one logical quote

EduMatcher keeps matching logic simple by posting both quote legs as regular limit orders. Each order stores:

- `origin=QUOTE`
- `quote_id=<shared-id>`

This means price-time matching stays unchanged while quote controls are handled in engine orchestration.

9.2 Startup Quote Seeding

On every engine startup, `_load_config()` reads each symbol's `market_maker_quotes` list and posts the defined bid/ask pairs into the order book **after restored GTC state has been reloaded** and before any gateway connection is accepted. This gives every book a guaranteed spread from the first moment of the trading day

while preserving prior-session time priority.

Each seeded quote entry defines:

- `gateway_id` (must be a configured `MARKET_MAKER` gateway)
- `bid_price`, `ask_price`
- `bid_qty`, `ask_qty`
- optional `tif` (default `DAY`), optional `quote_id`

If at least one `MARKET_MAKER` gateway is configured, the engine **requires** each symbol to define at least one quote seed — the config loader rejects the file otherwise.

9.2.1 Opting out: `require_mm_seed_quotes: false`

In a real market, this requirement is not a training-wheel restriction: even on a company's IPO day, an exchange lines up a designated market maker with an agreed opening quote before the first trade, precisely so the book is never empty. EduMatcher's default mirrors that — it is the realistic behaviour, not an artificial one.

EduMatcher is a teaching exchange, though, and one of the things worth teaching is what goes wrong *without* that guarantee: how wide the first trade's spread can be, how a thin book reacts to the first aggressive order, what a circuit breaker does when there is no reference price to measure against. To run that exercise you need a `MARKET_MAKER` gateway present (so a market maker can connect and quote once the session is under way) but **no** startup seed — a truly empty book at the opening bell.

Set the top-level flag:

```
require_mm_seed_quotes: false
```

to allow exactly that: a `MARKET_MAKER` gateway configured with zero `market_maker_quotes` entries for one, some, or all symbols. Everything else about seeding is unchanged — a symbol that *does* define quote seeds still has them validated and injected the normal way. `pm-config-gen --no-mm-seed-quotes` generates this shape (see The Configuration Workflow).

9.2.2 Exact startup sequence

```
1.  Engine parses config / allowlists and binds the main PULL/PUB sockets
2.  Restore data/book_stats.json  → last_buy_price / last_sell_price per symbol
3.  Restore data/gtc_orders.json  → re-inject GTC orders with original timestamps
4.  Restore data/gtc_combos.json  → rebuild parent-child combo maps
5.  Inject market_maker_quotes    → one quote (bid + ask) per configured entry
        • quote legs posted with origin=QUOTE, quote_id=<configured or generated>
        • matching is enabled — a GTC resting order could cross a seed quote here
        • trades and fill messages are published immediately
6.  Inject market_maker_combos    → seeded combo orders at top level
7.  Bind drop-copy PUB :5557 if available
8.  Publish initial book snapshots for all symbols
9.  Begin accepting gateway connections
```

For a clean first run, clear persisted files so no stale state competes with your seeds:

```
rm -f data/gtc_orders.json data/book_stats.json data/gtc_combos.json
poetry run pm-engine --verbose
```

The engine will print one line per injected quote:

```
[ENGINE] MM quote SEED-MM01-AAPL-1 AAPL bid=209.00x2000 ask=211.00x2000 gw=MM01
[ENGINE] Injected 3 market-maker quote(s) and 0 combo(s).
```

9.2.3 `gateway_id` is an accounting identity, not a connectivity check

The `gateway_id` field in a seed quote serves two purposes:

1. It records which market maker *owns* the quote (for fill reporting, kill-switch scope, and disconnect cleanup).
2. It must match a configured `MARKET_MAKER` FIX gateway — this is validated at load time.

It does not require that gateway to be connected. `_load_config()` bypasses the normal `_gateway_status()` connectivity check. Seed quotes enter the book the moment the engine starts, before any participant has dialled in.

This is intentional: the seeds represent the MM's *agreed opening liquidity*, not a live quote submission.

9.3 Fills Against Seed Quotes Before the MM Connects

Because seed quotes are live in the book before the MM gateway connects, it is possible for another participant's aggressive order to cross a seed quote during continuous session before the MM has dialled in.

9.3.1 What actually happens

Step	Event
1	Engine injects seed quote `MM-AAPL-1` for MM01 (MM01 not yet connected)
2	Session transitions to continuous (e.g. via scheduler or manual command)
3	TRADER01 submits a MARKET or aggressive LIMIT that crosses the seed bid or ask
4	Normal price-time matching fires — a **trade executes**
5	`_on_quote_leg_filled` is called for the filled leg
6	Based on `quote_refresh_policy` for MM01's gateway config:
	• `INACTIVATE_ON_ANY_FILL` (default): sibling leg cancelled, quote removed from QuoteIndex, `INACTIVE_*` status published to PUB socket
	• `INACTIVATE_ON_FULL_FILL`: only cancels if the leg was fully filled
	• `NEVER_INACTIVATE`: both legs stay in the book, fill reduces remaining quantity

Step	Event
7	The `INACTIVE_*` status message is sent to `quote.status.MM01` on the PUB socket — but MM01 is not connected and has no subscriber, so the message is **silently dropped**
8	Book continues in continuous session with reduced or no MM liquidity

9.3.2 Circuit breakers can still fire

Circuit breakers are implemented in the current engine. If a fill against a seed quote breaches the configured threshold ladder for that symbol:

- the symbol is marked halted
- all resting quotes for that symbol are cancelled
- the engine publishes `circuit_breaker.halt.<SYMBOL>`
- new orders and new quotes for that symbol are rejected until the breaker resumes

When the halt ends, the engine runs a symbol-local uncross before continuous matching resumes. So an unattended seed quote can do more than thin the book: it can be the trade that triggers a temporary halt.

9.3.3 What the MM must do on connect

When MM01 eventually connects, the engine does **not** automatically re-quote. The seeded quote may be:

- **Still active** — if no one crossed it. The MM can leave it or replace it by submitting a fresh `QUOTE` message (which cancels-and-replaces).
- **Partially filled** — sibling still resting with reduced quantity (only under `NEVER_INACTIVATE`). The MM should inspect its book state and re-quote.
- **Inactive / gone** — consumed by the inactivation policy. The book has no MM quote for this symbol. The MM **must** re-quote immediately.

The MM gateway should query book state on connect and submit a fresh quote for every symbol where no active quote exists:

```
QUOTE|SYM=AAPL|BID=209.80|ASK=210.20|BID_QTY=2000|ASK_QTY=2000|TIF=DAY
```

If the seed is still active this replace-and-cancel is safe — the existing quote is cancelled and the new one is registered atomically.

9.4 Gateway Commands

Submit or replace a quote:

```
QUOTE|SYM=AAPL|BID=209.80|ASK=210.20|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=MM-AAPL-1
```

Cancel active quote for one symbol:

```
QUOTE_CANCEL|SYM=AAPL
```

Trigger kill-switch for your gateway (optional symbol scope):

```
KILL
KILL|SYM=AAPL
```

9.5 Message Flow

Gateway to engine:

- `quote.new`
- `quote.cancel`
- `risk.kill_switch`
- `system.gateway_disconnect`

Engine to gateway:

- `quote.ack.<GW_ID>`
- `quote.status.<GW_ID>`
- `risk.kill_switch_ack.<GW_ID>`

Quote status values used by the engine:

- `ACTIVE`
- `INACTIVE_BID_FILLED`
- `INACTIVE_ASK_FILLED`
- `CANCELLED`

9.6 Refresh Policies

Each gateway can define quote refresh behavior in The Configuration Workflow:

- `INACTIVATE_ON_ANY_FILL`: any fill on one leg inactivates the quote
- `INACTIVATE_ON_FULL_FILL`: only full fill inactivates the quote
- `NEVER_INACTIVATE`: quote is not auto-inactivated by fills

When inactivation is triggered, the engine cancels the opposite leg and emits `quote.status.<GW_ID>`. The opposite leg is pulled at the moment of the fill, before anything else that order causes is matched: a stop order the fill triggers cannot trade against it.

If the filled leg is only partially executed, its unfilled remainder stays resting even though the quote is now inactive. It remains a quote-origin order: disconnect cleanup and `KILL` remove it along with active quote legs.

The same holds for a quote that crosses the book when it arrives. Its `quote.ack` and `quote.status ACTIVE` come first, then the fills. If the bid's fill inactivates the quote, the ask never reaches the book: it is reported as `order.cancelled` with `QUOTE_LEG_FILLED`, and the quote's last status is `INACTIVE_BID_FILLED`. If the ask fills, the bid that has already rested is cancelled the same way.

MM obligation enforcement is optional and configured per gateway:

- `enforce_mm_obligation: false` (default): quote spread/size checks are skipped
- `enforce_mm_obligation: true`: engine enforces:
  - `mm_max_spread_ticks`
  - `mm_min_qty`

9.7 Disconnect And Risk Controls

Gateway disconnect behavior is configurable per gateway:

- `CANCEL_QUOTES_ONLY` (the default for every gateway): on disconnect, all active quote legs and any resting remainder of an inactivated quote are cancelled. Regular resting orders are left.
- `CANCEL_ALL`: on disconnect, quotes **and** all resting non-quote orders are cancelled, including any remainder of an inactivated quote.
- `LEAVE_ALL`: disconnect is recorded but nothing is cancelled. Quotes and orders remain in the book unchanged.

`KILL` is an immediate risk control that cancels:

- Active quote legs and resting remainders of inactivated quotes
- Non-quote resting orders (optionally scoped to a single symbol)

The acknowledgement counts quote-origin orders under `QUOTES`, including inactivated quote remainders, and non-quote orders under `ORDERS`.

Seed quotes and disconnect

If the MM gateway was never connected in the first place, `_handle_gateway_disconnect` is never called and the seed quotes are **not** auto-cancelled on any disconnect event. They remain live until they are filled, explicitly quote-cancelled, cancelled by a kill-switch from that gateway, or removed by another engine event such as a circuit-breaker halt.
Glossary →

CHAPTER 10

Market Making

Learning objectives

After reading this page you will understand:

- What a market maker is and why exchanges rely on them
- How to submit a two-sided quote with the `QUOTE` command
- What happens to a quote when one side fills (inactivation)
- How `quote_refresh_policy` controls inactivation behaviour
- How MM obligations enforce minimum spread width and size
- Why Market-Maker Protection (MMP) is a data model today, not an enforced engine feature
- How `disconnect_behaviour` determines what happens to your quotes if your gateway drops

Prerequisites: The Configuration Workflow — you need a gateway configured with `role: MARKET_MAKER` before the engine accepts quotes. ALF Console — understand how to connect a gateway terminal.

10.1 What is a market maker?

In a real exchange, **market makers** are specialist participants who commit to continuously posting a two-sided market — a price at which they will buy (**bid**) and a price at which they will sell (**ask**). In return for this commitment, they often receive reduced transaction fees or regulatory benefits.

Their presence is what makes a market *liquid*: without market makers, a buyer who arrives when no seller is resting would have to wait indefinitely. Market makers ensure there is almost always a price to trade at.

EduMatcher models this with the `MARKET_MAKER` participant role, the `QUOTE` command (which posts a bid and ask as a linked pair), and an optional obligations framework that enforces spread and size constraints.

10.2 Configuring a market-maker gateway

A gateway must be assigned the `MARKET_MAKER` role in `engine_config.yaml` before the engine will accept quotes from it:

```
participants:
  - id: MM01
    description: Market maker
    role: MARKET_MAKER
    quote_refresh_policy: INACTIVATE_ON_ANY_FILL   # default
    disconnect_behaviour: CANCEL_QUOTES_ONLY        # default
    enforce_mm_obligation: true
    mm_max_spread_ticks: 10        # max spread in ticks (10 ticks = $0.10 for tick_size=0.01)
    mm_min_qty: 100                # minimum size on each side
```

A `TRADER` gateway that tries to send a `QUOTE` command will receive a rejection: `Quotes are only allowed for MARKET_MAKER participants`.

10.3 The QUOTE command

A quote is a single atomic operation that posts **one bid order and one ask order** under a shared `QUOTE_ID`. Both legs are ordinary `LIMIT` orders in the book — the only difference is that they are tracked together for lifecycle management.

```
QUOTE|SYM=AAPL|BID=149.90|ASK=150.10|BID_QTY=500|ASK_QTY=500
```

Optional fields:

Field	Default	Description
`QUOTE_ID=`	Auto-generated	Label for this quote pair; used in status events and cancel
`TIF=`	`DAY`	`DAY`, `GTC`, `ATO`, or `ATC` are all accepted (same `TIF` enum as `NEW`); `DAY`/`GTC` control whether the quote survives to the next session. Unlike `NEW`, the engine applies no session-phase check for `ATO`/`ATC` quotes.

10.3.1 Validation rules

The engine rejects a quote if:

Condition	Rejection reason
Gateway role is not `MARKET_MAKER`	`Quotes are only allowed for MARKET_MAKER participants`
`BID_PRICE >= ASK_PRICE`	`Quote requires bid_price < ask_price`
Either quantity ≤ 0	`Quote quantities must be positive`
Symbol is halted	`{SYMBOL} is halted — quotes rejected during circuit breaker halt`
Spread exceeds `mm_max_spread_ticks`	`Spread {n} ticks exceeds max {m}`
Either side below `mm_min_qty`	`Quote size must be >= {n}`

Condition	Rejection reason
Sending a new quote replaces any existing quote for the same (gateway, symbol) pair	*(no rejection; the existing quote is cancelled with an ordinary* `quote.status CANCELLED` *event — see Quote lifecycle below)*

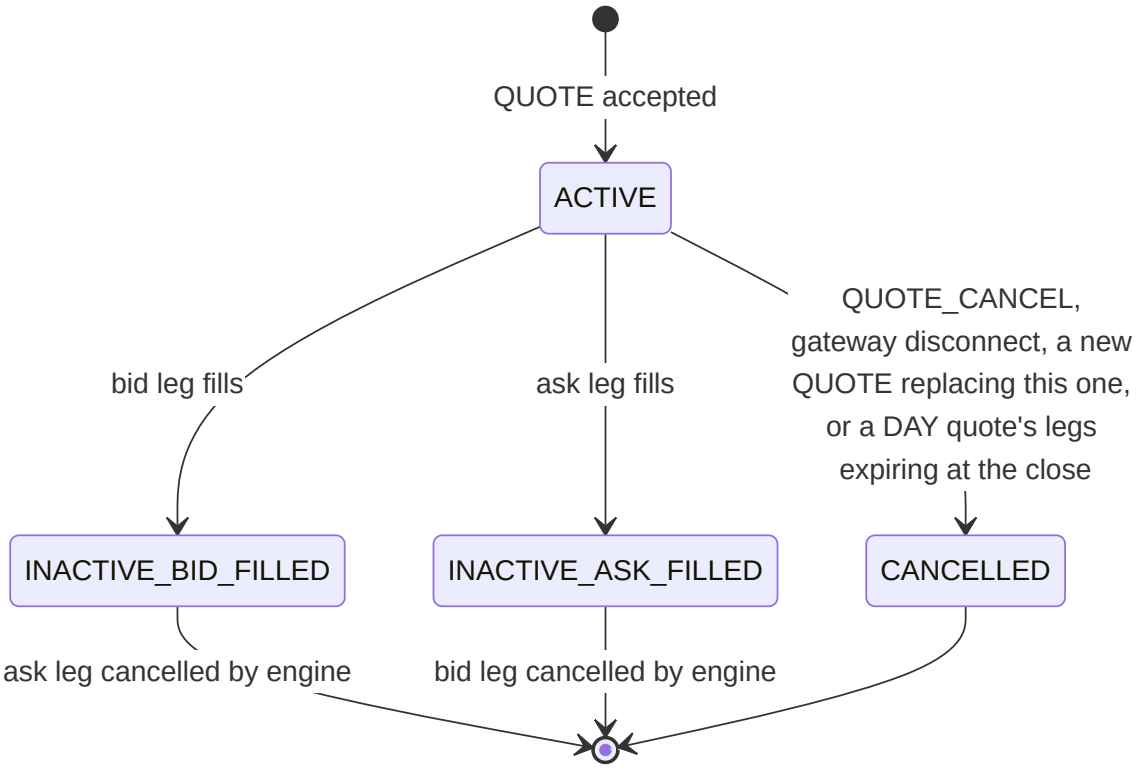
10.3.2 Quote acknowledgement

On success:

```
[HH:MM:SS] QUOTE ACK  q-aapl-001  bid=ord-aaa ask=ord-bbb
```

The `bid=` and `ask=` values are the individual order IDs assigned to each leg. These can be referenced individually in cancellation or via book queries.

10.4 Quote lifecycle



There is no separate `REPLACED` status: a new `QUOTE` on the same `(gateway_id, symbol)` cancels the old quote and the engine emits an ordinary `quote.status CANCELLED` for it (with reason `"Replaced by new quote"`) before installing the new one. Likewise there is no `EXPIRED` status: when a `TIF=DAY` quote's legs expire at the close (`order.expired` for each), the quote ends with `quote.status CANCELLED`, reason `"Expired"`. The engine's only `quote.status` values are `ACTIVE`, `INACTIVE_BID_FILLED`, `INACTIVE_ASK_FILLED`, and `CANCELLED`.

When inactivation occurs, the gateway receives a `quote.status` event:

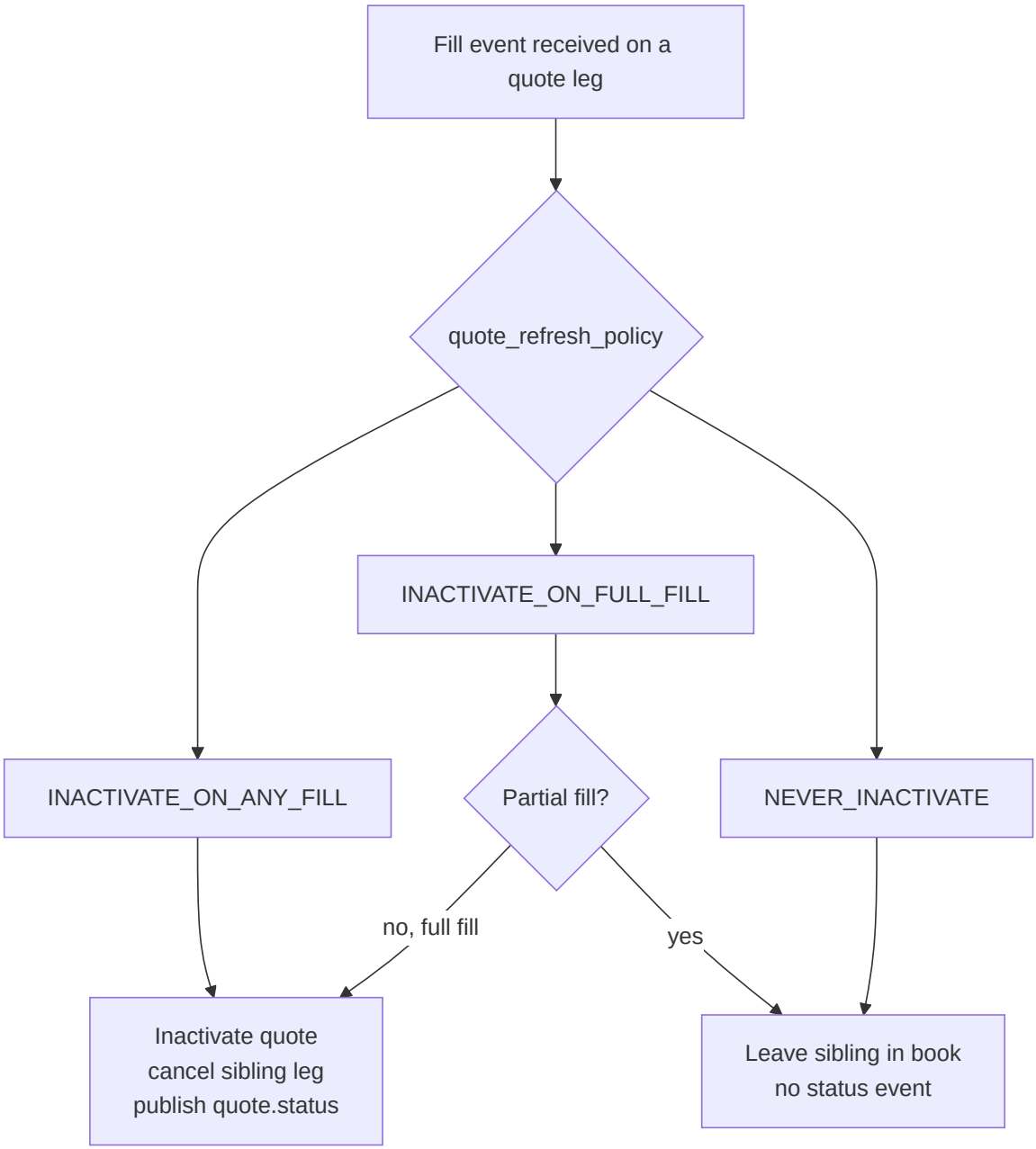
```
[HH:MM:SS] QUOTE INACTIVE_BID_FILLED  q-aapl-001
```

This tells the market maker: *"your bid was hit; your ask is now cancelled; you need to re-quote."*

10.5 Quote refresh policy

The `quote_refresh_policy` config key controls **what triggers inactivation**:

Policy	Inactivation trigger	Typical use
`INACTIVATE_ON_ANY_FILL`	Any partial or full fill on either leg	Conservative; re-quote after every fill event
`INACTIVATE_ON_FULL_FILL`	Only when a leg is **completely** filled	Allows partial fills to accumulate before re-quoting
`NEVER_INACTIVATE`	Never — both legs stay in the book until manually cancelled	High-volume automated MMs that manage their own inventory



NEVER_INACTIVATE plus GTC persistence is a durable steady-state across restarts

Under `NEVER_INACTIVATE`, a partially-filled leg is never cancelled by a fill — it just keeps resting with reduced quantity. Combined with quote persistence (this page, above), a `TIF=GTC` quote's remaining inventory now survives indefinitely across engine restarts, not just for one continuous run: restart the engine as many times as you like and a `NEVER_INACTIVATE` quote's partially-filled leg comes back with whatever quantity was left when it was last saved. This is very likely what you want if you're using `NEVER_INACTIVATE` to model an MM that manages its own inventory across sessions — but if you're using it for a classroom

exercise and expect each session to start from the full configured quantity, restart discipline alone will not reset it; delete `src/data/gtc_orders.json` (see the testing tip under `seed_once` above) or switch the leg's `tif` to `DAY` and let a business-day rollover clear it.

10.5.1 What happens to the *hit* leg itself, not just the sibling

The table and flowchart above describe what happens to the quote as a whole and to the **untouched sibling leg**. They deliberately say nothing about the **hit leg's own remaining quantity** on a *partial* fill, because the answer is the same across all three policies and is easy to get wrong by assuming the fill and the cancellation happen together. They do not.

A partial fill never cancels the leg that was hit — under any policy. The exchange's order book only removes a leg from its indexes once that leg reaches `remaining_qty == 0` (status `FILLED`). A `PARTIAL` fill leaves the hit leg resting at its original price with its reduced quantity, exactly like any ordinary partially-filled order — it is genuine, live, matchable liquidity, not a stale artifact. A second, independent taker can still trade against that same remainder before the MM does anything else. This is true whether or not `quote_refresh_policy` decides to inactivate the *quote* as a bookkeeping concept:

Policy	On a **partial** fill of the hit leg	When is the hit leg's remainder actually cancelled?
`INACTIVATE_ON_ANY_FILL`	Quote is inactivated immediately; **sibling** is cancelled; **hit leg's remainder keeps resting**	Only when the MM's next `QUOTE_CANCEL` or replacement `QUOTE` for this symbol arrives
`INACTIVATE_ON_FULL_FILL`	Quote stays active; no cancellation of either leg; hit leg's remainder keeps resting	Same as above — cancellation is never automatic until a full fill happens, or the MM acts
`NEVER_INACTIVATE`	Quote stays active; no cancellation of either leg; hit leg's remainder keeps resting	Only when the MM explicitly cancels or replaces

In other words: **the "inactivation" a policy decides on is a statement about the *sibling* leg and about whether the engine considers the quote slot still active — it is never a statement that the hit leg's own remainder disappears.** That remainder is cleaned up by exactly one mechanism, common to all three policies: the engine's **replace-in-slot logic**, which runs whenever a `QUOTE_CANCEL`, a new `QUOTE` or a startup seed quote arrives for that `(gateway_id, symbol)`. That logic cancels *every* order it can still find resting for this gateway's quote on this symbol — the untouched sibling if one is still there, and the hit leg's own partial remainder if one is still there — before installing the new quote (or, for `QUOTE_CANCEL`, before confirming the cancel). This holds even when the engine's own bookkeeping for "which quote was this" has already been cleared by an earlier fill-driven inactivation (`INACTIVATE_ON_ANY_FILL`): the replace logic still finds and cancels the leftover order by scanning the gateway's own resting quote legs directly, not only through that bookkeeping record. See Replace-by-new-quote case below for the exact mechanism and Case A for the operator-facing consequence.

Practical takeaway for every policy

Never assume "the quote is inactive" means "nothing from that quote is resting anymore." After any partial fill, run `QLEGS|SYM=<symbol>|SHOW=ALL` if you want to see exactly what is still on the book before you decide how much to quote next — or simply trust that your next `QUOTE` or `QUOTE_CANCEL` for that

symbol will clean up whatever is left, because the engine's replace-in-slot logic guarantees it.

10.6 MM obligations enforcement

When `enforce_mm_obligation: true` is set, the engine validates every `QUOTE` command against two constraints:

10.6.1 Spread constraint

$$\text{spread_ticks} = \frac{\text{ask_price} - \text{bid_price}}{\text{tick_size}}$$

The spread in ticks must not exceed `mm_max_spread_ticks`. Example: if `tick_size = 0.01` and `mm_max_spread_tick = 10`, then a quote of `BID=149.90 ASK=150.10` has a spread of **20 ticks** and will be **rejected**.

10.6.2 Minimum size constraint

Both `BID_QTY` and `ASK_QTY` must be `mm_min_qty`. Posting 50 on one side when `mm_min_qty = 100` will be **rejected**.

10.6.3 Per-symbol overrides

Obligations can be configured with four levels of precedence, from lowest to highest:

```
global mm_obligation_defaults (flat fields)
  └── per-gateway defaults (participants[].mm_max_spread_ticks, ...)
       └── per-symbol global policy (mm_obligation_defaults.symbols.<SYM>)
            └── per-gateway per-symbol policy (participants[].mm_obligations.<SYM>)
```

A per-gateway per-symbol override (`participants[].mm_obligations.<SYM>`) wins over everything else; a global per-symbol override (`mm_obligation_defaults.symbols.<SYM>`) wins over a gateway's own flat defaults. This lets you enforce tight spreads on liquid symbols while being more lenient on illiquid ones, all from a single config file.

Field names differ inside mm_obligations

Inside a gateway's `mm_obligations.<SYM>` block the size/spread fields are named `max_spread_ticks` and `min_qty` — **without** the `mm_` prefix used everywhere else in the config. This is intentional in the current schema, not a typo.
Example:

```
mm_obligation_defaults:
  enforce_mm_obligation: true
  mm_max_spread_ticks: 20
  mm_min_qty: 50

participants:
  - id: MM01
    role: MARKET_MAKER
    mm_obligations:
      AAPL:
        max_spread_ticks: 5    # tighter spread required on AAPL
        min_qty: 200
```

10.7 Market-Maker Protection (MMP) — data model only, not enforced by the engine

Real market makers are exposed to **adverse selection**: a rapid stream of fills on one side can mean an informed trader is hitting their price while they cannot adjust fast enough. A full Market-Maker Protection feature would provide an automatic pause when fill activity exceeds a threshold.

Not currently wired into the engine

`MarketMakerObligation` (`src/edumatcher/models/mm_obligation.py`) defines `mmp_fill_count` (default 5), `mmp_window_ns` (default 1,000,000,000 ns), and `max_requote_delay_ns` (default 500,000,000 ns) as dataclass fields, and a companion `MMPState` class implements the fill-counting and requote-deadline logic (`record_fill` / `activate_mmp` / `reset_mmp`). However, **none of this is read from `engine_config.yaml`** — `config_loader.py` never parses `mmp_fill_count`, `mmp_window_ns`, or `max_requote_delay_ns` from any config section — and **`MMPState` is never instantiated anywhere in `engine/main.py`**. There is no automatic pause, no requote deadline, and no "MMP breach" flag in the running engine today. A burst of fills on a quote leg behaves exactly like any other fill: it is governed only by `quote_refresh_policy` (see above). Treat this section as a description of the underlying data model that a future MMP feature could build on, not as documentation of live engine behaviour.

10.8 Cancelling a quote

```
QUOTE_CANCEL|SYM=AAPL
```

This cancels both legs atomically. The gateway receives a `quote.status` event with state `CANCELLED`.

10.9 Startup seeding — pre-loading quotes from config

A live market maker's first action after connecting is to post a quote. In a classroom or demo environment it is useful to have quotes already in the book *before any participant connects*, so the book is never completely empty and price discovery can begin from a known starting point.

market_maker_quotes is mandatory by default once any MARKET_MAKER gateway is configured

Config loading enforces this by default: if `participants` contains **any** gateway with `role: MARKET_MAKER` and the top-level `require_mm_seed_quotes` flag is left at its default (`true`), then **every** configured symbol must have at least one `market_maker_quotes` entry, or `load_engine_config()` raises `ValueError("Symbol '<SYM>': at least one market_maker_quotes entry is required when MARKET_MAKER gateways are configured (set require_mm_seed_quotes: false to allow an empty book)")` and the engine refuses to start. Set `require_mm_seed_quotes: false` at the top level of `engine_config.yaml` to opt out and allow a MARKET_MAKER-configured symbol to start with an empty book — see The Configuration Workflow for this flag. Each seed's `gateway_id` must also reference a gateway that is actually configured with `role: MARKET_MAKER` — a seed pointing at a `TRADER` gateway is rejected at load time too.

Seed quotes are the symbol's IPO opening quote

A `market_maker_quotes` seed is the day-one **opening quote** for a newly listed symbol — the exchange equivalent of an IPO. It should straddle the symbol's opening reference price (`last_buy_price` / `last_sell_price`) so the book and both risk-control references agree at the open. With `seed_once: true` (the default) the MM effectively "takes over" from day 2 onwards. See Configuration - Adding or Removing Symbols and Risk Controls - Day one (IPO) behaviour.
This is done with the `market_maker_quotes` key under each symbol in `engine_config.yaml`:

```
symbols:
  AAPL:
    tick_size: 0.01
    market_maker_quotes:
      - gateway_id: MM01      # must be a configured MARKET_MAKER gateway
        quote_id: SEED-MM01-AAPL
        bid_price: 149.95
        ask_price: 150.05
        bid_qty: 500
        ask_qty: 500
        tif: DAY              # or GTC — see below
        seed_once: true       # default — inject only on the very first startup
```

The engine injects these quotes during `_load_config()`, which runs at every startup after GTC orders are restored. Each entry creates a linked bid/ask pair in the order book exactly as if the market-maker gateway had sent a `QUOTE` command — the only difference is that no live gateway connection is needed.

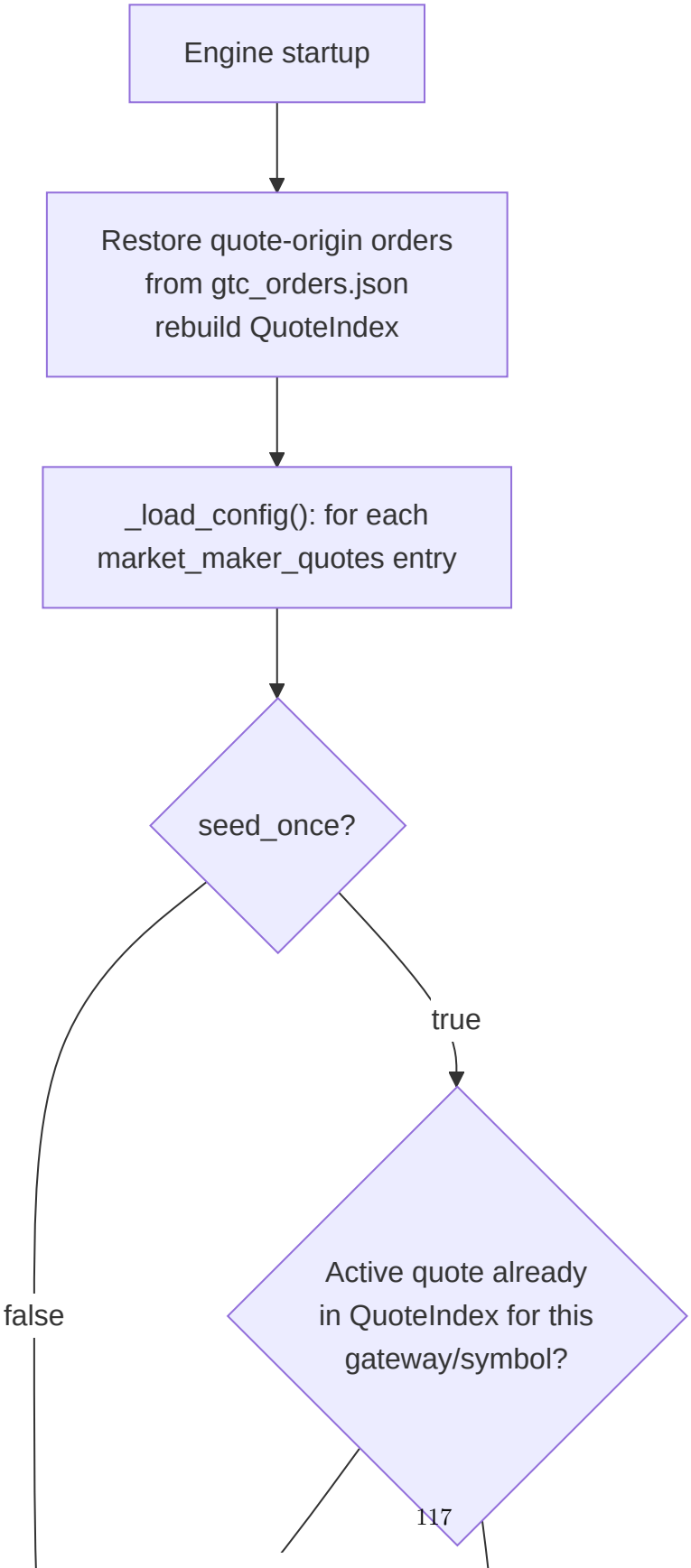
If the MM gateway later connects and sends a new `QUOTE` command for the same symbol, the seed is silently replaced (same behaviour as any re-quote).

10.9.1 Controlling when seeds are applied: `seed_once`

The `seed_once` field controls whether a seed is a **one-off primer** or a **permanent seed**:

`seed_once`	When is the seed injected?	Typical use
`true` *(default)*	Only when this gateway/symbol does **not** already have an active quote restored from a previous session — never again once a live quote for it exists	Initial liquidity for a brand-new symbol; the MM takes over once its quote is actually resting
`false`	On **every startup**, regardless of whether a quote already exists for this gateway/symbol	Demo setups where a specific spread must always be the opening quote

Detection is based on live quote presence, not on trading history: at startup the engine first restores any quote legs that survived the previous shutdown (see Cross-session behaviour below) and rebuilds its in-memory `QuoteIndex` from them, *before* deciding whether to inject config seeds. `seed_once: true` skips injection only when this specific `(gateway_id, symbol)` pair already has an active entry in that index — i.e. a quote is genuinely resting in the book right now. A symbol that has traded before, but whose quote later got fully hit through (or expired as a stale `TIF=DAY` order — see Choosing `tif` for seed quotes below), is correctly treated as having **no** active quote, so the seed fires again on the next startup rather than leaving the book with no MM presence.

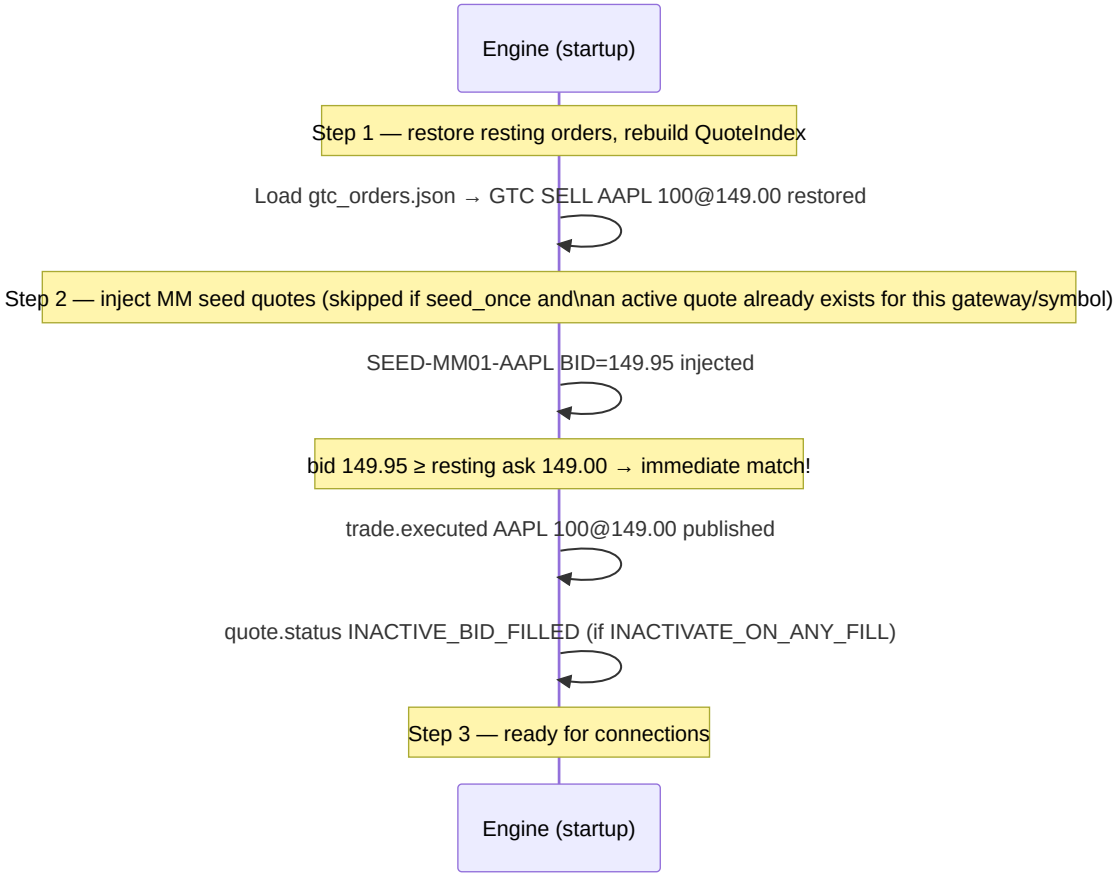

Engine startup
Restore quote-origin orders
from gtc_orders.json
rebuild QuoteIndex
_load_config(): for each
market_maker_quotes entry
seed_once?
true
false
Active quote already
in QuoteIndex for this
gateway/symbol?

Resetting to "first day" for testing

Delete `src/data/gtc_orders.json` (and, if you also want prior trade history and reference prices gone, `src/data/book_stats.json`) before starting the engine. With no persisted quote to restore, every `seed_once: true` seed fires again on the next startup — this is the standard way to reset a demo exchange to day-one state.

10.9.2 Seeding and the startup order

Seed quotes are injected *after* GTC/DAY resting orders — including any restored quote legs — are restored. This means a resting order from a previous session, GTC or DAY alike, may immediately cross against a seed quote — that trade fires at startup, before any gateway connects:



A restored quote leg can also cross a freshly-seeded quote from a *different* gateway — e.g. gateway A's restored quote only partially survived (§6.3 of `docs-design/EduMatcher-Revised-Quote-Persistence.md`), so `seed_once` still allows gateway B's seed to fire, and gateway B's seed happens to cross gateway A's surviving leg. This is the same startup-crossing behaviour as above, just between two market makers instead of a trader and a market maker.

Startup trades fire before any gateway is connected

If a GTC or same-day DAY sell order rests at a price that a seed bid would cross, the trade executes at startup. Fill events are published to the bus — `pm-clearing` and `pm-stats` record them — but no participant terminal is connected yet. The MM gateway's inbox will have the fill event waiting when it connects.

10.9.3 What happens on subsequent days?

A process exit — clean or otherwise — is **not** a day boundary. Quote legs persist across a restart by the same rule as any other order: `TIF=GTC` unconditionally, `TIF=DAY` only if the restart happens on the same business day it rested on (machine-local calendar date). The table below summarises the full session-boundary behaviour:

Component	Saved at shutdown?	Next-startup behaviour
GTC participant orders	Yes → `gtc_orders.json`	Restored into book before seed injection
GTC combo parent state	Yes → `gtc_combos.json`	Restored; parent-child links rebuilt
Book stats (OHLCV, last prices)	Yes → `book_stats.json`	Restored; used for `last_buy_price`/`last_sell_price`/`prev_close` only
DAY participant orders (same business day)	Yes → `gtc_orders.json`	Restored into book before seed injection
DAY participant orders (from a prior business day)	Yes → `gtc_orders.json` (until the next restart discards it)	Discarded at restore time, logged at `INFO`
MM quote legs (any `tif`, same business day if `DAY`)	Yes → `gtc_orders.json`, `QuoteIndex` rebuilt	Restored as a live, quote-managed quote — no re-seed if `seed_once: true`
MM seed quotes (`seed_once: true`, no active quote restored)	—	Injected fresh
MM seed quotes (`seed_once: false`)	—	Re-injected on every startup regardless of what was restored

Note that a symbol's `book_stats.json` entry no longer has anything to do with whether a `seed_once: true` seed fires — see Controlling when seeds are applied above. It continues to drive `last_buy_price`/`last_sell_price`/`prev_` and the price-collar/circuit-breaker reference price, which is unrelated to quote seeding.

Quote legs are persisted like any other order

Earlier versions of this engine excluded quote-origin orders from `gtc_orders.json` entirely, and expired every `TIF=DAY` order (quotes included) at every shutdown. Neither is true any more: a quote leg persists

and restores exactly like a participant's order would, subject to the same `TIF=GTC`-unconditional / `TIF=DAY`-same-business-day rule. See `docs-design/EduMatcher-Revised-Quote-Persistence.md` (repository checkout only) for the full design rationale, and Persistence for the general GTC/DAY persistence mechanics this relies on.

10.9.4 Choosing `tif` for seed quotes

`tif` value *(default `DAY`)*	Behaviour during the session	Cross-restart behaviour
`DAY`	Quote expires at end of trading day (scheduler-driven `CLOSED`, not a process exit): `order.expired` for each leg, then `quote.status` `CANCELLED` (reason `"Expired"`)	Survives a restart within the same business day; discarded at restore if the business day has rolled over
`GTC`	Quote survives an ATC/session reset within the same engine run	Survives every restart unconditionally, regardless of business day

Both values now behave the same as an ordinary participant order at the same `tif` — quotes are no longer a special case for persistence. Choose `DAY` (the default) for a quote that should reset with the trading day like a real IPO opening quote; choose `GTC` only if you deliberately want the seed's resting inventory to carry over indefinitely across business-day boundaries without being re-primed from config.

10.10 Disconnect behaviour

If a market maker's gateway disconnects (Ctrl-C or network failure), the engine must decide what to do with any resting quotes:

`disconnect_behaviour`	What happens to quotes	What happens to plain orders
`CANCEL_QUOTES_ONLY`	Active quotes and inactive partial-fill remainders are cancelled	Plain resting orders remain in book
`CANCEL_ALL`	Active quotes and inactive partial-fill remainders are cancelled	All plain resting orders are also cancelled
`LEAVE_ALL`	Nothing cancelled	Nothing cancelled

The default is `CANCEL_QUOTES_ONLY`, which is the most appropriate for market making: quotes represent ongoing commitment and should not linger after the MM disconnects. A quote may become inactive after one leg partially fills while that leg's unfilled remainder still rests; disconnect cleanup treats that remainder as a quote and cancels it under either `CANCEL_QUOTES_ONLY` or `CANCEL_ALL`. Participant `KILL` also cancels it, and reports it in the `QUOTES` count.

10.11 Full worked example

Scenario: `MM01` provides a continuous two-sided market in `AAPL`. `GW02` is a customer who buys 200 shares.

```
# Configure engine_config.yaml:
#   gateways.MM01.role = MARKET_MAKER
#   gateways.MM01.enforce_mm_obligation = true
#   gateways.MM01.mm_max_spread_ticks = 10
#   gateways.MM01.mm_min_qty = 100
#   gateways.MM01.quote_refresh_policy = INACTIVATE_ON_ANY_FILL

# 1. MM01 posts a quote
[MM01]> QUOTE|SYM=AAPL|BID=149.95|ASK=150.05|BID_QTY=500|ASK_QTY=500|QUOTE_ID=q1
[14:30:00] QUOTE ACK  q1  bid=ord-001 ask=ord-002

# 2. GW02 buys 200 at market — hits the ASK leg
[GW02]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=200
[14:30:01] FILL      ord-003  qty=200 @150.05  remaining=0  [FILLED]

# MM01 sees:
[14:30:01] FILL      ord-002  qty=200 @150.05  remaining=300  [PARTIAL]  (fill on ask leg)
[14:30:01] QUOTE INACTIVE_ASK_FILLED  q1          (bid leg auto-cancelled)

# 3. MM01 re-quotes
[MM01]> QUOTE|SYM=AAPL|BID=149.95|ASK=150.05|BID_QTY=500|ASK_QTY=300|QUOTE_ID=q2
[14:30:01] QUOTE ACK  q2  bid=ord-004 ask=ord-005
```

After the customer fill, `MM01`'s bid leg (`ord-001`) was cancelled by the engine automatically. The market maker posted a new quote `q2` with 300 on the ask to reflect the inventory consumed.

10.12 Config reference summary

```
participants:
  - id: MM01
    role: MARKET_MAKER
    quote_refresh_policy: INACTIVATE_ON_ANY_FILL   # or INACTIVATE_ON_FULL_FILL / NEVER_INACTIVATE
    disconnect_behaviour: CANCEL_QUOTES_ONLY        # or CANCEL_ALL / LEAVE_ALL
    enforce_mm_obligation: true
    mm_max_spread_ticks: 10
    mm_min_qty: 100
    mm_obligations:              # per-symbol overrides (optional); note: no mm_ prefix here
      TSLA:
        max_spread_ticks: 20
        min_qty: 50
```

10.13 MM quote identification and quote-leg mapping

10.13.1 Scope

The rest of this chapter explains exactly how EduMatcher represents a market-maker quote, how a quote is mapped to its two child orders, how the MM can identify that a fill belongs to the currently active quote, and what the MM must do to cancel or re-issue a quote.

It answers these questions:

1. Is a quote identified by `quote_id`, or by `(gateway_id, symbol)`?
2. How is a quote mapped to the resting order(s)?
3. How does the MM learn that one quote leg was taken?
4. When is sibling cancellation automatic, and when must the MM do it?
5. What state and data must the MM keep in order to re-quote safely?

The chapter describes the engine behavior in full details

10.13.2 Executive summary

Identity model

EduMatcher uses two identifiers for a quote, and they serve different roles:

- Active quote slot in the engine: `(gateway_id, symbol)`
- External correlation identifier: `quote_id`

Practical meaning:

- The engine allows at most one active quote slot per `(gateway_id, symbol)`.
- A new `QUOTE` on the same gateway and symbol replaces the previous active slot.
- `quote_id` identifies the specific logical quote instance occupying that slot.

So the answer is:

- The engine routes and replaces quotes by `(gateway_id, symbol)`.
- The MM should correlate business events by `quote_id`.

Mapping model

Each quote becomes two ordinary limit orders:

- bid leg: `side=BUY`
- ask leg: `side=SELL`

The mapping is explicit in both directions:

- `QuoteIndex[(gateway_id, symbol)] -> QuoteEntry(quote_id, bid_order_id, ask_order_id)`
- each leg order stores `origin=QUOTE` and `quote_id=<same quote_id>`

How the MM identifies a fill as belonging to the active quote

The fill arrives as a normal `order.fill.<gateway_id>` event for a leg `order_id`.

Important implementation detail:

- `order.fill` **does** include `quote_id` in its published payload (an `Order`'s `quote_id` field is carried straight through to the fill event).

Therefore the MM must identify quote-leg fills by keeping the mapping returned by `quote.ack`:

- `quote_id -> {symbol, bid_order_id, ask_order_id}`
- `order_id -> {quote_id, leg_side, symbol}`

Then the rule is simple:

- if `fill.order_id == bid_order_id`, the fill belongs to the bid leg of that quote
- if `fill.order_id == ask_order_id`, the fill belongs to the ask leg of that quote

When cancellation is automatic

Automatic **sibling** cancellation depends on `quote_refresh_policy`:

- `INACTIVATE_ON_ANY_FILL`: sibling leg is auto-cancelled on any fill
- `INACTIVATE_ON_FULL_FILL`: sibling leg is auto-cancelled only when the filled leg reaches `remaining_qty=0`
- `NEVER_INACTIVATE`: no automatic sibling cancellation due to fills

The `quote_refresh_policy` is set per gateway in `engine_config.yaml` under `participants[].quote_refresh_policy`.

This is about the sibling leg — not the leg that was actually hit

None of these three policies ever automatically cancels the *hit* leg's own remaining quantity on a partial fill; that remainder rests, live and tradeable, until the MM's own next `QUOTE`/`QUOTE_CANCEL` for that symbol replaces it. See What happens to the hit leg itself, not just the sibling for the full explanation and a policy-by-policy table.

What the MM needs in order to re-issue a quote

To re-issue a quote, the MM needs:

- the symbol
- the new bid price and ask price
- the new bid quantity and ask quantity
- the desired `TIF`
- a new or reused client-side quote label strategy for `QUOTE_ID`
- the current quote mapping so fills and cancels can be correlated correctly

In practice the MM should keep:

- current active quote per symbol
- reverse map from child `order_id` to `quote_id`
- local state showing whether the quote is active, cancelled, or inactivated
- local pending-submission state for edge cases where a fill arrives before `quote.ack`

10.13.3 Exact implementation model

10.13.4 Quote identity: `quote_id` vs `(gateway_id, symbol)`

Both are used, but they are not interchangeable.

`(gateway_id, symbol)` is the engine's active-slot key

Internally the engine stores active quotes in `QuoteIndex`, keyed by:

- `gateway_id`
- `symbol`

Consequences:

- there can be only one active quote slot per gateway and symbol
- `QUOTE_CANCEL` targets the active slot by symbol, not by `quote_id`
- a new `QUOTE` on the same symbol replaces the currently indexed slot

`quote_id` is the MM's logical correlation key

`quote_id` is:

- optional on submission
- auto-generated by the engine if omitted
- returned in `quote.ack`
- carried in `quote.status`
- copied into both child orders internally

The engine therefore treats `quote_id` as the identifier for the specific quote generation, but not as the routing key for cancel/replace.

10.13.5 Quote-to-leg mapping

Engine-side mapping

The engine stores one `QuoteEntry` per active quote slot:

- `quote_id`
- `gateway_id`
- `symbol`
- `bid_order_id`
- `ask_order_id`

That gives deterministic lookup from quote slot to its two child orders.

Order-side mapping

Each child order stores:

- `origin=QUOTE`
- `quote_id=<quote_id>`

That gives deterministic lookup from child order back to the originating quote.

What the MM actually receives

The MM does **not** receive the full internal `Order` object on `order.fill`. The fill payload includes fields such as:

- `order_id`

- `fill_qty`
- `fill_price`
- `remaining_qty`
- `status`
- `symbol`
- `side`
- `order_type`
- `tif`
- `qty`
- `price`

It **does** include `quote_id` (see above).

So the MM must not assume that `order.fill` alone is enough to identify the logical quote instance unless the MM already persisted the order-id mapping from `quote.ack`.

10.13.6 Quote lifecycle messages and what they mean

`quote.ack.<gateway_id>`

Purpose: acceptance or rejection of a quote request.

Fields:

- `quote_id`
- `accepted`
- `reason`
- `bid_order_id`
- `ask_order_id`

Important details:

- on rejection, `accepted=false` and no leg IDs are usable
- on acceptance, `bid_order_id` and `ask_order_id` are the critical correlation keys
- on explicit `QUOTE_CANCEL`, the engine currently sends an acceptance ack with the `quote_id`, but no leg IDs

`quote.status.<gateway_id>`

Purpose: quote lifecycle transition.

Current status values emitted by the engine:

- `ACTIVE`
- `INACTIVE_BID_FILLED`
- `INACTIVE_ASK_FILLED`
- `CANCELLED`

Important detail:

- `REJECTED` is **not** a `quote.status` value in the engine
- rejection is reported only through `quote.ack accepted=false`

order.fill.<gateway_id>

Purpose: execution detail for one child order.

This is the message that tells the MM that one quote leg traded.

order.cancelled.<gateway_id>

Purpose: cancellation detail for one child order.

This is how the MM learns that the sibling leg was actually cancelled as an order-book event.

10.13.7 Real message ordering in the engine

This is the most important correctness section in the note.

The engine does **not** always emit messages in the intuitive order `quote.ack -> quote.status ACTIVE -> later fill -> later inactive/cancel`.

Normal resting-quote case

If neither leg matches immediately on insert, the MM will typically see:

1. `quote.ack accepted=true`
2. `quote.status ACTIVE`
3. later, if a leg trades, one or more `order.fill`
4. if policy inactivates the quote, `order.cancelled` for the sibling leg
5. `quote.status INACTIVE_*`

Immediate-match-on-insert edge case

The engine inserts the two quote legs into the books **before** it emits `quote.ack` and `quote.status ACTIVE`.

Therefore, if one leg matches immediately on entry, the MM may observe:

1. `order.fill` for the filled leg
2. possibly `order.cancelled` for the sibling leg
3. `quote.status INACTIVE_*`
4. only then `quote.ack accepted=true`
5. and then `quote.status ACTIVE`

That ordering is counter-intuitive, but it is what the current engine code does.

Practical consequence for the MM:

- the MM must tolerate fills arriving before `quote.ack`
- the MM must be able to reconcile those early fills once `quote.ack` provides `bid_order_id` and `ask_order_id`

Explicit cancel case

On `QUOTE_CANCEL|SYM=<symbol>`, the engine currently does:

1. cancel any still-resting bid/ask child orders, emitting `order.cancelled` per resting child
2. emit `quote.status CANCELLED`
3. emit `quote.ack accepted=true`

So in this path, `quote.status CANCELLED` comes **before** the final successful `quote.ack`.

Replace-by-new-quote case

On a new `QUOTE` for the same `(gateway_id, symbol)`, the engine looks up the active `QuoteIndex` entry for that slot. What happens next depends on whether one is still there:

If an active `QuoteIndex` entry is found (the ordinary case — no fill has inactivated this quote since it was last (re)issued):

1. the engine removes that `QuoteIndex` entry
2. cancels the old quote's two tracked legs (whichever of them are still resting — one may already be gone from an earlier fill or cancel)
3. emits old-quote `quote.status CANCELLED`, with per-leg fill/cancel detail attached
4. creates new legs
5. eventually emits new-quote `quote.ack accepted=true`
6. emits new-quote `quote.status ACTIVE`

If no active `QuoteIndex` entry is found — most commonly because `INACTIVATE_ON_ANY_FILL` already inactivated this quote at fill time (see the table above) — there is nothing left in the `QuoteIndex` bookkeeping to replace, but there can still be a genuinely resting order on the book: the hit leg's own partial remainder, which that earlier fill deliberately left in place. The engine handles this with a fallback: it looks up the gateway's own resting quote-origin orders on this symbol directly (via `OrderBook`'s per-gateway index of resting quote legs, not through the `QuoteIndex`) and cancels anything it finds there too, **before** creating the new legs. See docs/architecture/020-guided-tour.md §10 for exactly how that index is kept in sync and why it exists alongside `QuoteIndex` rather than replacing it. No `quote.status CANCELLED` is emitted for this fallback cancellation — the quote was already announced `INACTIVE_BID_FILLED`/`INACTIVE_ASK_FILLED` at fill time, so there is no quote-level status left to transition — but an ordinary `order.cancelled` is still published for the cancelled leg, exactly as for any other engine- initiated cancellation. Then:

1. creates new legs
2. eventually emits new-quote `quote.ack accepted=true`
3. emits new-quote `quote.status ACTIVE`

Either way, the practical guarantee for the MM is the same: **sending a new `QUOTE` for a symbol you already have a quote on always leaves at most that one new quote's two legs resting for your gateway on that symbol** — never a leftover from whatever came before, regardless of which policy or which fill pattern produced that leftover.

Again, the lifecycle is not simply "ack first, status second" across all paths.

10.13.8 ALF interaction examples

10.13.9 Example 1: normal quote, later bid-leg fill, automatic sibling cancel

Assume:

- gateway: `MM01`
- symbol: `AAPL`
- policy: `INACTIVATE_ON_ANY_FILL`
- MM chooses `QUOTE_ID=Q123`

MM submits quote

```
[MM01]> QUOTE|SYM=AAPL|BID=209.80|ASK=210.20|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=Q123
```

Engine accepts and returns mapping

Gateway output will look like:

```
[09:30:00] QUOTE ACK   Q123  bid=B1A8C2D4 ask=S9F3E1AA
[09:30:00] QUOTE ACTIVE  Q123
```

At this point the MM must persist:

- `Q123 -> {symbol=AAPL, bid_order_id=B1A8C2D4, ask_order_id=S9F3E1AA, tif=DAY}`
- `B1A8C2D4 -> {quote_id=Q123, leg=BID, symbol=AAPL}`
- `S9F3E1AA -> {quote_id=Q123, leg=ASK, symbol=AAPL}`

Later the bid leg is hit

Suppose another participant sells 100 into the MM's 500-quantity bid — a **partial** fill, chosen deliberately for this example because it is the case that is easy to get wrong (a full fill collapses the nuance below, since there is no remainder left to reason about). The MM may see:

```
[09:31:02] FILL      B1A8C2D4  qty=100 @209.80  remaining=400  [PARTIAL]
[09:31:02] CANCELLED  S9F3E1AA
[09:31:02] QUOTE INACTIVE_BID_FILLED  Q123
```

How the MM identifies the fill as belonging to the active quote

The fill correlation logic is:

1. read `order.fill.order_id`
2. look it up in `order_id -> quote_id`
3. conclude that `B1A8C2D4` belongs to `Q123`
4. because it is the stored `bid_order_id`, conclude that the bid leg of `Q123` was taken

Does the MM need to cancel the sibling leg?

No, not in this policy.

Under `INACTIVATE_ON_ANY_FILL`, the engine already did both of these things:

- removed the quote from the active quote index
- cancelled the sibling ask leg (`S9F3E1AA`) automatically

The `order.cancelled` and `quote.status INACTIVE_BID_FILLED` messages are the observable confirmation of that automatic cleanup.

But notice what is conspicuously absent from that list: `B1A8C2D4` itself — the leg that was actually hit. Its `remaining=400` is not cancelled by this fill. It stays resting on the book, at `209.80`, as genuine tradeable liquidity, for as long as the MM takes to react — a slow MM, or one that never re-quotes at all, leaves that 400 quantity live indefinitely. This is correct, documented engine behavior, not a bug: `INACTIVATE_ON_ANY_FILL`'s job is to protect the MM from the untouched *sibling* leg going stale, not to instantly zero out a leg's own inventory the moment it starts trading. The MM does not need to send anything to clean up `B1A8C2D4` — sending its replacement `QUOTE` (next section) does that automatically as

a side effect of the engine's replace-in-slot logic, described in Replace-by-new-quote case above. Until that replacement `QUOTE` arrives, though, `B1A8C2D4`'s remaining 400 keeps trading exactly like any other resting order.

10.13.10 Example 2: full-fill-only policy

Assume policy `INACTIVATE_ON_FULL_FILL`.

If the bid leg is only partially filled, the MM may see:

```
[09:31:02] FILL      B1A8C2D4  qty=100 @209.80  remaining=400  [PARTIAL]
```

And that may be all.

In that case:

- no sibling auto-cancel occurs yet
- no `quote.status INACTIVE_*` is emitted yet
- the quote remains active in the engine

The MM therefore continues to treat the quote as active until either:

- the filled leg becomes fully filled, causing inactivation, or
- the MM explicitly cancels or replaces the quote

10.13.11 Example 3: `NEVER_INACTIVATE`

Assume policy `NEVER_INACTIVATE`.

The MM may see:

```
[09:31:02] FILL      B1A8C2D4  qty=100 @209.80  remaining=400  [PARTIAL]
```

And no automatic sibling cancel, and no `INACTIVE_*` status.

Meaning:

- the quote slot remains active in the engine
- the sibling ask leg remains resting unless something else removes it
- the MM must decide whether to keep quoting, cancel, or replace

If the MM wants a fresh two-sided quote immediately, it has two valid options:

Option A: explicit cancel then new quote

```
[MM01]> QUOTE_CANCEL|SYM=AAPL
```

Typical resulting events:

```
[09:31:02] CANCELLED  S9F3E1AA
[09:31:02] QUOTE CANCELLED  Q123  Cancelled by participant
[09:31:02] QUOTE ACK  Q123
```

Then the MM submits a new quote.

Option B: direct replacement quote

```
[MM01]> QUOTE|SYM=AAPL|BID=209.70|ASK=210.10|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=Q124
```

The engine will:

- remove the current active quote slot for `(MM01, AAPL)`
- cancel any surviving old child orders
- emit `quote.status CANCELLED` for the old quote
- install the new quote

This means the MM does **not** have to send `QUOTE_CANCEL` first if it is immediately replacing the quote with another quote on the same symbol.

10.13.12 Example 4: immediate fill before `quote.ack`

This is the tricky case that every robust MM must handle.

Suppose the quote is marketable as soon as it is inserted. The MM may see:

```
[09:30:00] FILL      B1A8C2D4  qty=500 @209.80  remaining=0  [FILLED]
[09:30:00] CANCELLED  S9F3E1AA
[09:30:00] QUOTE INACTIVE_BID_FILLED  Q123
[09:30:00] QUOTE ACK  Q123  bid=B1A8C2D4 ask=S9F3E1AA
[09:30:00] QUOTE ACTIVE  Q123
```

Yes, that ordering is possible in the current engine.

What the MM must do in this case

The MM needs one more local concept:

- `pending_quote_by_symbol`

When it sends:

```
QUOTE|SYM=AAPL|...|QUOTE_ID=Q123
```

it should locally remember that there is a pending quote submission for AAPL.

Then if an early fill arrives before `quote.ack`, the MM can:

1. temporarily buffer the fill by `(gateway_id, symbol)`
2. wait for `quote.ack`
3. use `bid_order_id` and `ask_order_id` from the ack to retroactively resolve that buffered fill
4. then continue normal state handling

Without this pending-submit buffer, the MM cannot deterministically map an early fill to the quote until the ack arrives.

10.13.13 MM operator workflow via `pm-alf-console`

This section is intentionally operator-oriented. It shows what a market maker using the interactive ALF gateway will actually type, what the gateway will print back, and what the operator must remember while managing live quotes.

10.13.14 What the operator sees on screen

The gateway displays quote and order lifecycle events in a compact terminal format.

Important display behavior:

- `QUOTE ACK` shows the `quote_id` and the full child order ID for each leg
- `FILL` shows the full filled child order ID
- `CANCELLED` shows the full cancelled child order ID
- `QUOTE INACTIVE_*` and `QUOTE CANCELLED` show the `quote_id`

The gateway prints full order IDs (32-character hex strings) — it does not truncate them. So a human operator correlates events using:

- `quote_id` for the logical quote instance
- the full order IDs for the bid and ask legs (shortened to 8 characters in the examples below purely for readability)

To reduce manual correlation load in fast markets, the gateway supports:

```
QLEGS[|SYM=<symbol>][|SHOW=ACTIVE|RECENT|ALL]
```

`QLEGS` shows per-gateway quote legs with explicit `Filled` and `Filled?` columns, so the operator can identify the traded leg without mentally joining multiple event lines.

QLEGS reads a local cache, not the engine

`QLEGS` is answered entirely from `quote_leg_cache`, a table the gateway builds up locally from `quote.ack`, `order.fill`, and `order.cancelled` events it has received **during the current connection**. It does not send any request to the engine. If the gateway just (re)connected and missed earlier events, or if `SHOW=RECENT`/`SHOW=ALL` is requested, `QLEGS` can only show what this session has observed — it is not a guaranteed-authoritative snapshot of engine state.
For an authoritative, engine-side view of currently active quotes, use `QBOOT` instead:

```
QBOOT[|SYM=<symbol>]
```

`QBOOT` sends a `system.quote_bootstrap_request` to the engine and prints the reply (`system.quote_bootstrap.<gateway_id>`), which is built directly from the engine's own `QuoteIndex` — the same `QuoteEntry` records described earlier in this chapter. Each row shows `quote_id`, `state`, `bid`/`ask` price, and `BidRem`/`AskRem` (remaining quantity per leg). This is the right tool to run right after connecting or reconnecting, to recover the current `quote_id` → leg mapping without having witnessed the original `quote.ack`.
A programmatic MM should keep the full IDs from the message payload. A human operator using the terminal sees those same full IDs printed — the shortened IDs in the examples below (e.g. `7c4a91e2`) stand in for the full 32-character order ID for readability.

10.13.15 Typical manual quoting session

Assume the MM operator is connected as `MM01` and wants to quote `AAPL`.

Step 1: submit a quote

Operator input:

```
[MM01]> QUOTE|SYM=AAPL|BID=209.80|ASK=210.20|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=Q123
```

Typical gateway output when the quote rests normally:

```
[09:30:00.101] QUOTE ACK   Q123  bid=7c4a91e2 ask=be2170fd
[09:30:00.102] QUOTE ACTIVE  Q123
```

What the operator must remember immediately

At this moment the operator should record or mentally associate:

- symbol: `AAPL`
- quote id: `Q123`
- bid leg prefix: `7c4a91e2`
- ask leg prefix: `be2170fd`
- intended prices: `209.80 / 210.20`
- intended sizes: `500 / 500`
- current refresh policy for that gateway

The minimum safe correlation set for a human operator is:

- `Q123`
- bid leg order ID
- ask leg order ID

10.13.16 Step 2: identify which leg filled

Later the operator may see:

```
[09:31:02.417] FILL      7c4a91e2  qty=100 @209.8  remaining=400  [PARTIAL]
```

The operator identifies this as the quote bid leg because:

- the filled order-id prefix `7c4a91e2` matches the bid ID shown in `QUOTE ACK`
- therefore the bid side of `Q123` traded

If instead the operator saw:

```
[09:31:02.417] FILL      be2170fd  qty=100 @210.2  remaining=400  [PARTIAL]
```

that would mean the ask side of `Q123` traded.

With `QLEGS`, this becomes a direct read:

```
[MM01]> QLEGS|SYM=AAPL|SHOW=ALL
```

Interpretation pattern:

- `Filled?=YES` with `Leg=BUY` means the bid leg traded
- `Filled?=YES` with `Leg=SELL` means the ask leg traded
- `Rem>0` with leg status `NEW` or `PARTIAL` identifies still-active exposure

10.13.17 Step 3: interpret what happens next

What the operator should do next depends on the configured refresh policy.

Case A: `INACTIVATE_ON_ANY_FILL`

Typical terminal output:

```
[09:31:02.417] FILL      7c4a91e2  qty=100 @209.8  remaining=400  [PARTIAL]
[09:31:02.418] CANCELLED be2170fd
[09:31:02.418] QUOTE INACTIVE_BID_FILLED  Q123
```

Operator interpretation:

- the bid leg traded, but only partially — `7c4a91e2` still has 400 quantity resting at its original price, live and tradeable, right now
- the engine automatically cancelled the sibling ask leg (`be2170fd`)
- the quote is no longer active as a bookkeeping concept, but `7c4a91e2`'s remaining 400 is **not** cancelled by this — it keeps resting until the operator's next `QUOTE` or `QUOTE_CANCEL` for this symbol arrives
- the operator may now prepare and submit a replacement quote

Operator action:

- no manual `QUOTE_CANCEL` is needed — the next `QUOTE` for this symbol will cancel `7c4a91e2`'s leftover 400 automatically as part of its ordinary replace-in-slot handling, alongside installing the new legs
- compute the new bid/ask and send a fresh `QUOTE`
- optionally run `QLEGS|SYM=AAPL|SHOW=ALL` before re-quoting to confirm final leg state

Case B: `INACTIVATE_ON_FULL_FILL`

Typical terminal output after a partial fill:

```
[09:31:02.417] FILL      7c4a91e2  qty=100 @209.8  remaining=400  [PARTIAL]
```

Operator interpretation:

- one quote leg traded
- the quote may still be active
- no sibling cancellation has happened yet
- the quote stays live until full fill or explicit operator action

Operator action choices:

- leave the quote working if that is desired
- send `QUOTE_CANCEL|SYM=AAPL` if the quote should be withdrawn now
- send a replacement `QUOTE` if the operator wants to replace the current quote immediately

Case C: `NEVER_INACTIVATE`

Typical terminal output:

```
[09:31:02.417] FILL      7c4a91e2  qty=100 @209.8  remaining=400  [PARTIAL]
```

Operator interpretation:

- the bid leg traded
- the quote remains active unless manually changed or removed by another engine event
- the sibling ask leg is still live

Operator action choices:

- do nothing and keep the surviving structure live
- cancel explicitly with `QUOTE_CANCEL|SYM=AAPL`
- replace directly by sending a new `QUOTE` on `AAPL`

10.13.18 Explicit cancel workflow from the terminal

If the operator wants to cancel the current active quote for `AAPL`:

```
[MM01]> QUOTE_CANCEL|SYM=AAPL
```

Typical output:

```
[09:31:20.010] CANCELLED 7c4a91e2
[09:31:20.011] CANCELLED be2170fd
[09:31:20.011] QUOTE CANCELLED  Q123  Cancelled by participant
[09:31:20.012] QUOTE ACK  Q123
```

Important operator note:

- `QUOTE_CANCEL` is symbol-based, not `quote_id`-based
- it targets whatever quote is currently active in the `(gateway_id, symbol)` slot

So before typing `QUOTE_CANCEL|SYM=AAPL`, the operator should be sure that the currently active AAPL quote is the one they intend to remove.

10.13.19 Direct replacement workflow from the terminal

In many cases the operator does not need a separate cancel command. The engine already supports replace-by-new-quote semantics.

Example:

```
[MM01]> QUOTE|SYM=AAPL|BID=209.70|ASK=210.10|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=Q124
```

Typical interpretation:

- if `Q123` was still the active AAPL quote, the engine will remove it first
- any still-resting old legs are cancelled
- old quote lifecycle ends with `QUOTE CANCELLED  Q123`
- new leg IDs are allocated for `Q124`
- the operator then receives a new `QUOTE ACK` for `Q124`

This is often the cleanest manual workflow for active MM operation.

10.13.20 What the operator must remember after every `QUOTE ACK`

After every accepted quote, the operator should refresh their working state.

The operator should remember:

- current symbol being quoted
- current `quote_id`
- bid leg order ID
- ask leg order ID
- whether the current policy auto-inactivates on any fill, only full fill, or never
- whether a fill already happened and whether the quote is still active

If the operator loses track of which leg IDs belong to which quote, then later `FILL` and `CANCELLED` lines become ambiguous.

10.13.21 Operator fill-handling checklist

When a `FILL` line appears, the operator should follow this sequence:

1. Match the displayed order-id prefix against the latest `QUOTE ACK` for that symbol.
2. Determine whether the bid leg or ask leg traded.

3. Check `remaining=` and `[status]` on the fill line.
4. Watch immediately for either:
   - `CANCELLED` on the sibling leg
   - `QUOTE INACTIVE_BID_FILLED`
   - `QUOTE INACTIVE_ASK_FILLED`
5. Decide whether the quote is now inactive or still live.
6. If inactive, compute and submit the next quote.
7. If still live, decide whether to leave it working, cancel it, or replace it.

10.13.22 What is needed by the MM to re-issue a quote

To re-issue a usable two-sided quote, the operator or MM logic must know:

- `SYM`
- new `BID`
- new `ASK`
- new `BID_QTY`
- new `ASK_QTY`
- `TIF`
- new `QUOTE_ID` if the MM uses explicit client-side quote labels

Typical re-quote command:

```
[MM01]> QUOTE|SYM=AAPL|BID=209.75|ASK=210.15|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=Q124
```

Operationally, the MM also needs to know whether the old quote is actually gone.

Safe trigger points for re-quoting are:

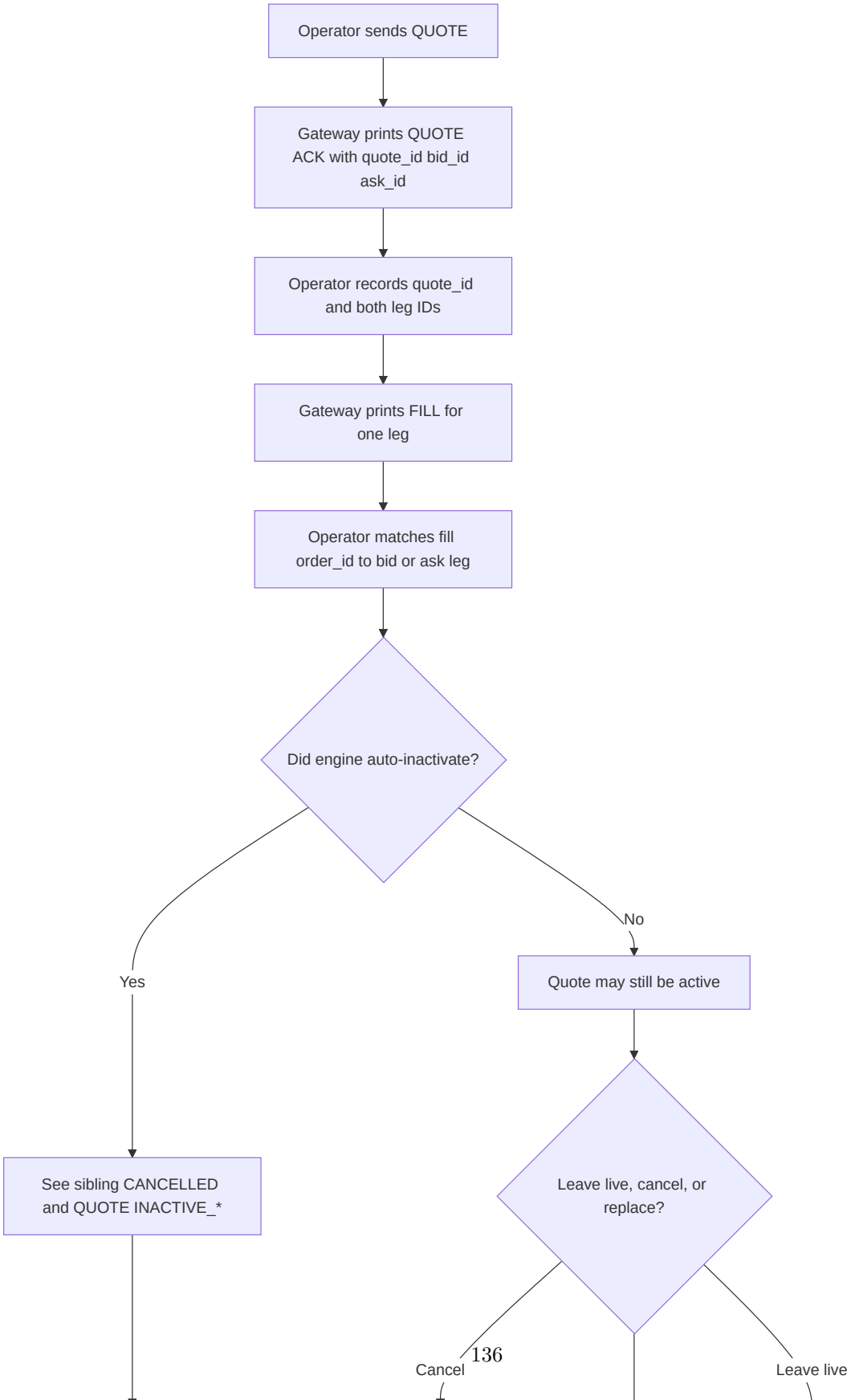
- after `QUOTE INACTIVE_*`
- after `QUOTE CANCELLED`
- after a deliberate replace-by-new-quote decision

10.13.23 Recommended operator habit

For manual terminal operation, the safest habit is:

1. always supply your own `QUOTE_ID`
2. after each `QUOTE ACK`, note the bid and ask leg order IDs
3. after each `FILL`, identify which leg traded by matching the order ID
4. wait for `QUOTE INACTIVE_*` if the gateway auto-inactivates
5. submit the next `QUOTE` only once you know whether the old quote is still active

This reduces confusion during fast markets and makes post-trade reconciliation much easier.


Operator sends QUOTE
Gateway prints QUOTE ACK with quote_id bid_id ask_id
Operator records quote_id and both leg IDs
Gateway prints FILL for one leg
Operator matches fill order_id to bid or ask leg
Did engine auto-inactivate?
Yes
No
See sibling CANCELLED and QUOTE INACTIVE_*
Quote may still be active
Leave live, cancel, or replace?
Cancel
Leave live

10.13.24 Recommended MM-side local state

The MM should maintain these structures.

Active quote record per symbol

```
active_quote[symbol] = {
  quote_id,
  bid_order_id,
  ask_order_id,
  bid_price,
  ask_price,
  bid_qty,
  ask_qty,
  tif,
  state,
}
```

Reverse index by order ID

```
order_to_quote[order_id] = {
  quote_id,
  symbol,
  leg_side,   ## BID or ASK
}
```

Pending submit state

```
pending_quote[symbol] = {
  client_quote_id,
  bid_price,
  ask_price,
  bid_qty,
  ask_qty,
  tif,
  sent_at,
}
```

This exists specifically to survive the immediate-fill-before-ack edge case.

10.13.25 What the MM must know to re-issue a quote

To issue a new quote, the MM must compute and send:

- `SYM`
- `BID`
- `ASK`
- `BID_QTY`
- `ASK_QTY`
- optional `TIF`
- optional `QUOTE_ID`

Example:

```
[MM01]> QUOTE|SYM=AAPL|BID=209.70|ASK=210.10|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=Q124
```

What the MM should normally do before re-issuing

If policy auto-inactivates Recommended sequence:

1. receive `order.fill` for the quote leg
2. receive `quote.status INACTIVE_*` or otherwise confirm old quote is no longer active
3. compute the new two-sided quote
4. send the new `QUOTE`

Manual `QUOTE_CANCEL` is usually unnecessary here.

If policy does not auto-inactivate The MM must choose one of:

1. send `QUOTE_CANCEL|SYM=<symbol>`, wait for cancellation lifecycle, then send a new `QUOTE`
2. send a replacement `QUOTE` directly on the same symbol, relying on the engine's replace semantics

What the MM must not assume

The MM must not assume:

- that `order.fill` contains `quote_id`
- that `quote.ack` always arrives before every other quote-related message
- that `quote.status ACTIVE` always means the quote is still active at the moment it is received
- that `QUOTE_CANCEL` is keyed by `quote_id` rather than by symbol

10.13.26 Recommended local state machine

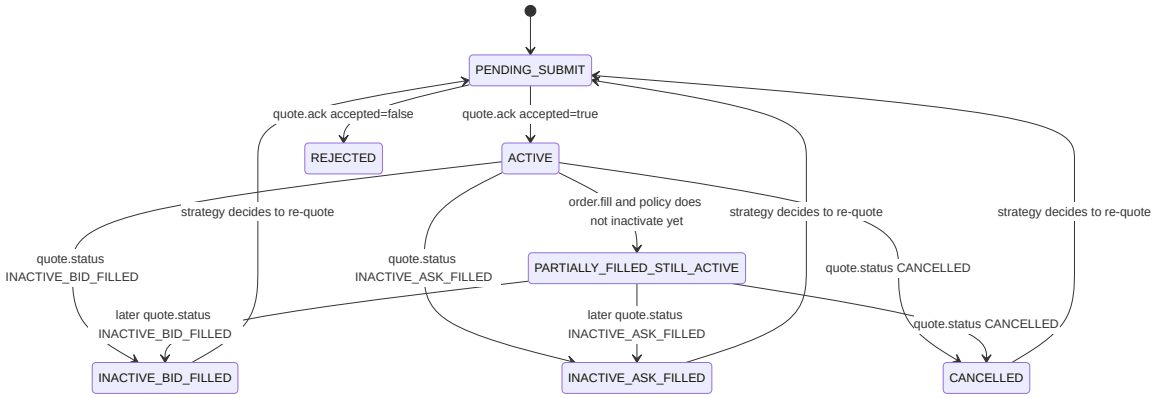
The engine publishes only a few quote lifecycle statuses. The MM usually needs a slightly richer local state machine.

Recommended local states:

- `PENDING_SUBMIT`
- `ACTIVE`
- `PARTIALLY_FILLED_STILL_ACTIVE`
- `INACTIVE_BID_FILLED`
- `INACTIVE_ASK_FILLED`
- `CANCELLED`
- `REJECTED`

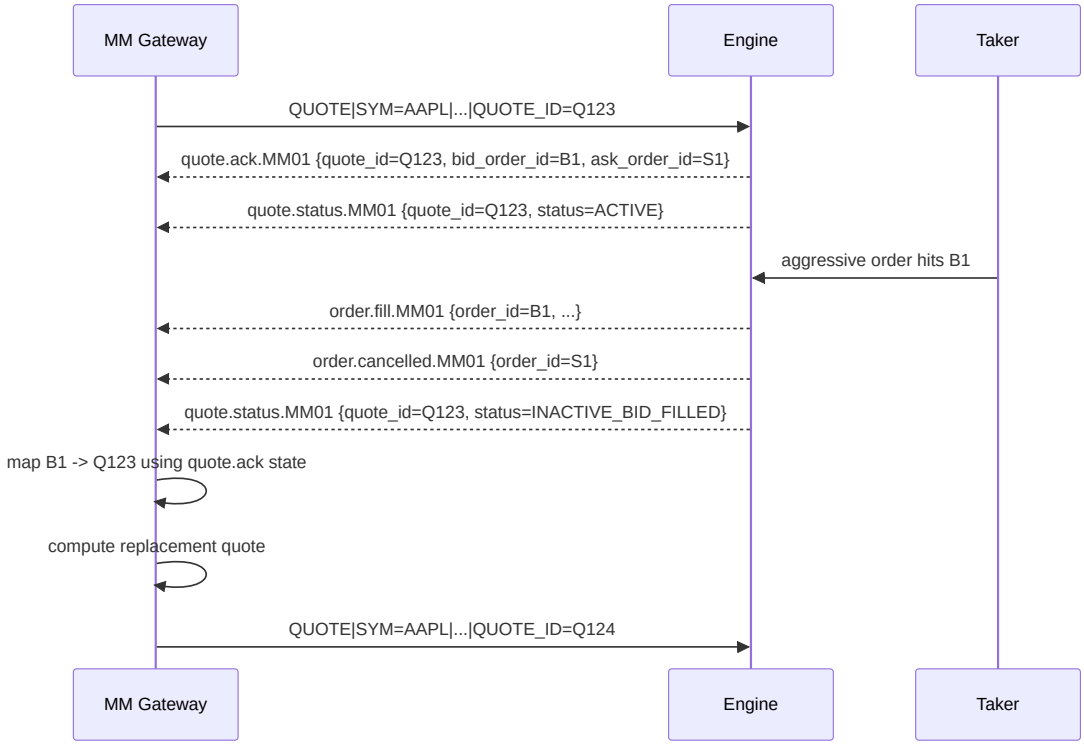
Notes:

- `PARTIALLY_FILLED_STILL_ACTIVE` is an MM-local state, not an engine `quote.status`
- `REJECTED` is also MM-local; the engine reports rejection only through `quote.ack accepted=false`



10.13.27 Sequence graph: normal auto-inactivate flow

This is the cleanest sequence to reason about and the best default example for docs or code comments.



10.13.28 Practical conclusions

- The engine's unique active quote slot is `(gateway_id, symbol)`.
- `quote_id` identifies the logical quote instance occupying that slot.
- The MM identifies quote-leg fills by `order_id`, not by `quote_id` in the fill payload.
- `quote.ack` is the critical mapping message because it provides `bid_order_id` and `ask_order_id`.
- Under `INACTIVATE_ON_ANY_FILL`, sibling cancellation is automatic.

- Under `INACTIVATE_ON_FULL_FILL`, sibling cancellation happens only after full fill.
- Under `NEVER_INACTIVATE`, the MM must decide when to cancel or replace.
- **No policy ever automatically cancels the *hit* leg's own remainder on a partial fill, under any policy** — sibling cancellation and quote inactivation are statements about the untouched leg and the bookkeeping slot, not about the traded leg's own resting quantity. The hit leg's remainder is cleaned up by exactly one mechanism regardless of policy: the engine's replace-in-slot logic, triggered by the MM's own next `QUOTE` or `QUOTE_CANCEL` for that symbol.
- Re-quoting can be done either by explicit `QUOTE_CANCEL` followed by new `QUOTE`, or by sending a replacement `QUOTE` directly on the same symbol — either path guarantees any leftover resting quantity from the previous quote (sibling or hit-leg remainder alike) is cancelled first.
- The MM must tolerate edge cases where `order.fill` and even `quote.status INACTIVE_*` arrive before `quote.ack`.

10.13.29 Operational checklist for the MM implementation

- Always send your own `QUOTE_ID` for clean reconciliation.
- Persist `quote.ack` mappings immediately.
- Maintain `order_id -> quote_id` reverse lookup.
- Correlate fills by `order_id`, never by assumed `quote_id` in the fill payload.
- Subscribe to all of:
  - `order.fill.<gateway_id>`
  - `order.cancelled.<gateway_id>`
  - `quote.ack.<gateway_id>`
  - `quote.status.<gateway_id>`
- Add pending-submit buffering for the immediate-fill-before-ack edge case.
- Treat `quote.status INACTIVE_*` or `quote.status CANCELLED` as the authoritative signal that the previous quote slot is no longer active — **not** as a signal that every previously resting order from that quote is gone; a partially-filled leg can still be resting after either status.
- Use direct replacement `QUOTE` when you want the engine to perform atomic replace semantics on the same symbol — this is also what cancels any leftover partially-filled leg from the previous quote, whether or not `quote.status` ever reported it as inactive.

10.14 See also

In this book:

- The Market-Maker Bot (`pm-mm-bot`) — autonomous quoting process that implements the strategies described on this page
- Order Types — the LIMIT orders that quote legs create under the hood
- ALF Console — full QUOTE and QUOTE_CANCEL command syntax

Operator's Guide:

- Configuration — full gateway and `mm_obligation_defaults` config schema
- Risk Controls — how circuit breakers cancel quotes during a halt
- Persistence — GTC and same-day DAY quotes survive restarts; MM seeds are injected at startup for whatever didn't restore

Protocols and Clients:

- Messages — `quote.ack` and `quote.status` message payloads

CHAPTER 11

The Market-Maker Bot (`pm-mm-bot`)

Learning objectives

After reading this page you will understand:

- What `pm-mm-bot` is and how it differs from manual `QUOTE` commands
- How to launch one or more autonomous market-maker bot instances
- How one instance can quote several symbols at once with `--symbols`, and when to prefer that over one process per symbol
- The bot's lifecycle: startup handshake, quoting, repricing, and shutdown
- How quote refresh works after fills and mid-price drift
- How the `passive` strategy lets other traders trade first, stepping behind their prices and backing off after a fill
- How to configure gateway entries in `engine_config.yaml` for MM bots
- How to bootstrap a fresh exchange with no existing book data
- How to keep a bot's parameters in a version-controlled `--config` file

Prerequisites: Market Making — understand the `QUOTE` command, `quote_refresh_policy`, and `disconnect_behaviour` before using the bot. The Configuration Workflow — each bot instance needs a pre-registered `MARKET_MAKER` gateway in the engine config.

11.1 What is pm-mm-bot?

`pm-mm-bot` is an **autonomous market-maker process** that keeps one or more symbols liquid without human intervention. It connects to the engine as a single `MARKET_MAKER` gateway, posts a two-sided quote (bid and ask) per symbol it covers, and automatically reprices each one when:

- One side of that symbol's quote is filled
- That symbol's mid-price drifts beyond a configurable threshold
- The session state changes (e.g. entering or leaving an auction phase)

A single instance can quote **one symbol or several** from the same process, behind the same gateway ID — nothing on the engine side ties a `MARKET_MAKER` gateway to a single symbol (the engine's `QuoteIndex` keys every active quote by `(gateway_id, symbol)` and tracks a *set* of such keys per gateway). Each symbol progresses

through its own copy of the bot's state machine independently: a fill, cancel, drift, or QLEGS divergence on one symbol never touches another symbol's quoting.

```
# One process, one symbol (the classic form, still the default)
pm-mm-bot --symbol AAPL

# One process, several symbols sharing one set of settings
pm-mm-bot --symbols AAPL,MSFT,TSLA

# One process, several symbols, each on its own terms
pm-mm-bot --symbol AAPL --gap 0.10 \
          --symbol MSFT --gap 0.20 --qty 200 \
          --symbol TSLA --gap 0.50 --qty 100 --strategy inventory_skew --max-position 5000

# The same thing, from a file
pm-mm-bot --config mm_tech.yaml
```

Running one instance per symbol (as separate processes) is still fully supported and is the right choice when you want independent OS-level failure domains — a crash in one symbol's process cannot affect another's:

```
pm-mm-bot --symbol AAPL &
pm-mm-bot --symbol MSFT &
pm-mm-bot --symbol TSLA &
```

Multiple instances can also compete on the same symbol — they appear as independent market makers in the book (use `--id-suffix` to distinguish them).

Feature	Manual `QUOTE`	`pm-mm-bot`
Requires human operator	Yes	No
Automatic reissue after fill	No	Yes
Drift-based repricing	No	Yes
Session-aware (pause in auctions)	Manual	Automatic
Multiple symbols per process	N/A	Built-in (`--symbol` repeated, or `--symbols`)
Different settings per symbol	N/A	Built-in (see Per-symbol settings)
Multiple instances per symbol	Possible	Built-in (`--id-suffix`)

One process vs. several — which to use

Both are legitimate, for different scenarios. One process is the right default: it is the only way to get a genuinely shared process (one PUSH/SUB connection, one auth handshake, one gateway identity) across symbols, and each symbol can still be tuned individually — spread, size, strategy and timers are all per-symbol settings. One process per symbol is the right choice when you want a separate **OS-level** failure domain per symbol, or when the scenario specifically calls for several independently-attributable market-maker identities (for example when teaching self-match prevention). Note that a multi-symbol bot already isolates a *failing* symbol without a separate process — see Per-symbol failure isolation.

11.2 Quick start

```
# Start a market maker for AAPL with default settings
pm-mm-bot --symbol AAPL

# With explicit spread and quantity
pm-mm-bot --symbol AAPL --gap 0.10 --qty 500

# In Poetry development mode
poetry run pm-mm-bot --symbol AAPL --gap 0.10 --qty 500 -v
```

Before launching, ensure the gateway ID is registered in `engine_config.yaml`:

```
participants:
  - id: MM_AAPL_01
    description: "AAPL market-maker bot"
    role: MARKET_MAKER
    disconnect_behaviour: CANCEL_QUOTES_ONLY
    quote_refresh_policy: INACTIVATE_ON_ANY_FILL
```

11.3 Gateway identity convention

A single-symbol bot instance uses the gateway ID format:

```
MM_<SYMBOL>_<nn>
```

Where `<SYMBOL>` is the symbol in uppercase and `<nn>` is the two-digit suffix from `--id-suffix` (default `01`).

`--symbol`	`--id-suffix`	Gateway ID
`AAPL`	`01` (default)	`MM_AAPL_01`
`AAPL`	`02`	`MM_AAPL_02`
`MSFT`	`01`	`MM_MSFT_01`

A `--symbols` bot derives the same shape from every symbol it covers, joined with underscores, unless `--label` overrides that segment directly:

`--symbols`	`--label`	Gateway ID
`AAPL,MSFT`	*unset*	`MM_AAPL_MSFT_01`
`AAPL,MSFT,TSLA`	`TECH`	`MM_TECH_01`

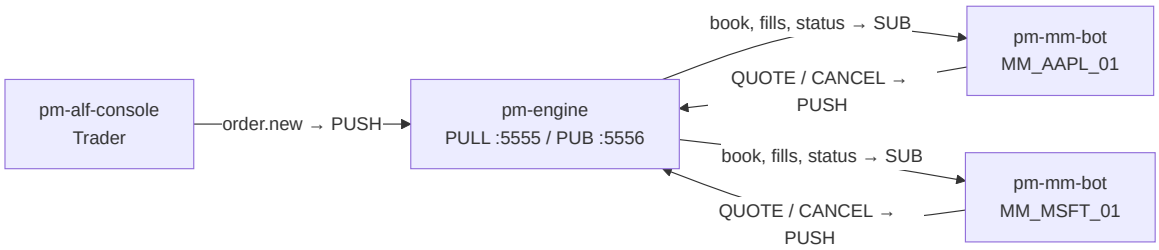
`--all-symbols` defaults the label to `ALL`, giving `MM_ALL_01`, so the gateway entry in `engine_config.yaml` does not change when symbols are added to or removed from the configuration. `--label` and `--gateway-id` still override it.

`--label` exists because `MM_<SYMBOL>_<nn>` has no natural multi-symbol form once the symbol list grows — a five-symbol gateway ID built by joining every symbol is legal but unwieldy in logs and `pm-admin`. Pick a short label (e.g. a sector or desk name) once the symbol list stops being self-describing at a glance.

This convention makes bot gateways immediately identifiable in logs, the admin console (`pm-admin`), and the order book viewer (`pm-board`).

11.4 Architecture

Each `pm-mm-bot` instance is a standalone process that communicates with the engine using the same ZMQ PUSH/SUB pattern as all other participants — one PUSH/SUB pair per process regardless of how many symbols that process quotes:

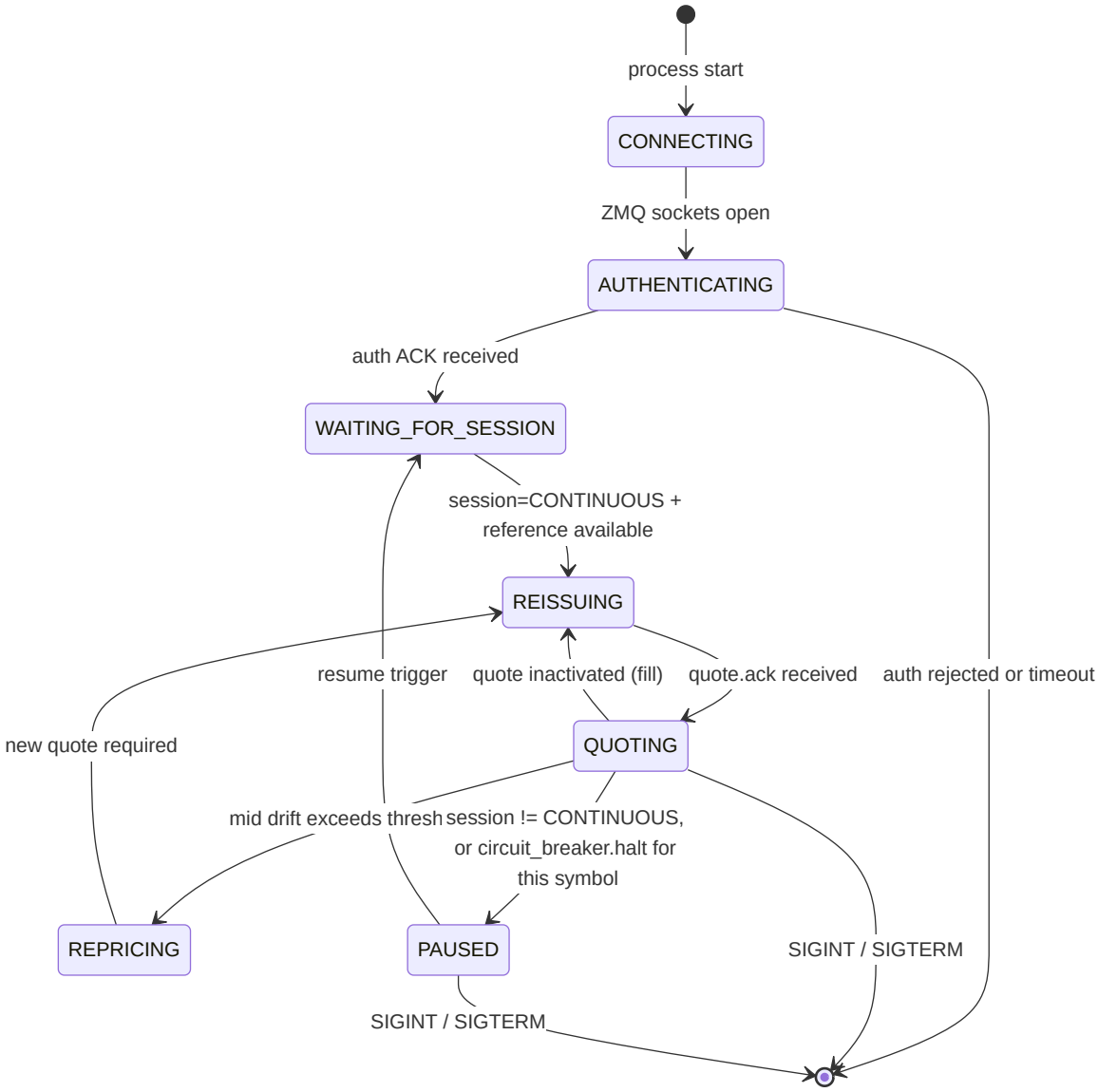


Here `B1` and `B2` are two separate processes, each quoting one symbol — the same diagram describes a single `--symbols AAPL,MSFT` process just as well by collapsing `B1`/`B2` into one box `MM_AAPL_MSFT_01` with two QUOTE streams into the same PUSH socket instead of two sockets.

11.5 Bot lifecycle

11.5.1 State machine

The bot progresses through a well-defined set of states:



11.5.2 Startup sequence

1. Open ZMQ sockets (PUSH and SUB) — once per process, not once per symbol
2. Send `gateway_connect` and wait for `gateway_auth` ACK — once per process
3. Request symbol list and verify every `--symbol`/`--symbols` entry exists
4. For each symbol: send `QBOOT` — if an active quote already exists for this `(gateway_id, symbol)` pair, adopt it instead of creating a duplicate
5. For each symbol: send `QLEGS` to reconcile quote-leg mapping
6. Wait for `session.state` event (fail fast if not received within timeout) — once per process, since session state applies to the whole exchange rather than to one symbol
7. For each symbol still active after steps 3–5: resolve its initial reference price and begin quoting

Steps 3–5 and 7 run once per symbol and are where per-symbol failure isolation applies: a symbol that fails

one of these checks is excluded from quoting rather than aborting the whole process, as long as at least one other symbol succeeds.

11.5.3 Per-symbol failure isolation

A `--symbols` bot does not treat a startup problem with one symbol as fatal to the others. If, say, `AAPL`'s `--gap` violates its `mm_max_spread_ticks` obligation while `MSFT`'s does not, the bot logs `AAPL` as excluded and continues quoting `MSFT` — the process only exits with a startup failure if *every* symbol fails its checks (exactly the single-symbol behavior, applied to the whole symbol set rather than to one symbol). The same isolation applies to any other per-symbol startup failure: an unknown symbol, an invalid strategy/gap/tick combination, or no reference price available.

This is a startup-time check only — once a symbol is quoting, a runtime problem specific to that symbol (a rejected quote, a circuit-breaker halt) already only affects that symbol's own state, the same way it always has; see Session state handling and Quote refresh logic above.

11.5.4 Graceful shutdown

On `SIGINT` (Ctrl+C) or `SIGTERM`, the bot:

1. Sends `quote.cancel` for every symbol still quoting
2. Waits up to `--shutdown-timeout-sec` for cancel confirmation
3. Closes ZMQ sockets and exits

11.6 Pricing logic

11.6.1 Mid-price tracking

The bot tracks the current mid-price from the order book:

- **Both sides present**: `mid = (best_bid + best_ask) / 2`
- **Ask only**: `mid = best_ask`
- **Bid only**: `mid = best_bid`
- **No data**: keep previous mid

The `symmetric` and `inventory_skew` strategies apply this to the whole book, including the bot's own quote. The `passive` strategy applies it to other traders' orders only — see The passive strategy.

11.6.2 Quote placement

Given the mid-price and `--gap` (total spread), the bot places:

- **Bid** at `mid - gap/2`, rounded to the nearest tick
- **Ask** at `mid + gap/2`, rounded to the nearest tick

A minimum spread of 2 ticks is always guaranteed, even after rounding.

This is the `symmetric` placement. `passive` treats it as the *home* price and may quote further out — see The passive strategy.

11.6.3 Drift detection

After posting a quote, the bot records the mid at the time of posting. On each book update, it checks whether the mid has moved by more than `--drift-ticks` ticks. If so, it cancels and reissues at the new mid. (`passive` uses its own, per-side re-quote rules — see When the bot re-quotes.)

11.7 Quote refresh logic

11.7.1 Refresh triggers

Trigger	Action
Quote inactivated (one side filled)	Reissue after `--reissue-delay-ms`
Mid-price drift exceeds threshold	Cancel active quote, then reissue at new mid
`passive` only: a side becomes/stops being covered, must step back, or a fade starts/ends	Cancel active quote, then reissue (see When the bot re-quotes)
Quote rejected	Retry after delay
Periodic heartbeat (no active quote)	Reissue
Periodic QLEGS reconciliation mismatch	Clear local state and reissue

11.7.2 Reissue delay

After a fill, the bot waits `--reissue-delay-ms` (default: 200 ms) before reissuing. If multiple fills arrive in quick succession, the timer resets on each fill — resulting in exactly one reissue after the burst settles.

11.7.3 Cancel timeout guard

When the bot is replacing an active quote, it first sends `quote.cancel` and waits up to `--cancel-timeout-sec` for lifecycle confirmation. If no confirmation arrives within that window, it forces a safe replacement by clearing local quote IDs and sending a fresh `quote.new`.

11.7.4 Self-healing against dropped messages

ZMQ delivery is best-effort, so the bot is built to recover if an engine reply is ever lost:

- **Dropped `quote.ack`** — after sending a quote the bot waits for the ack to confirm it. If the ack never arrives, the heartbeat guard notices it holds no live quote and, once a full `--heartbeat-interval-sec` has elapsed since the last quote was sent (so an in-flight ack is never pre-empted), it reissues.
- **Periodic QLEGS reconciliation** — every `--qlegs-reconcile-interval-sec` the bot requests a fresh quote-leg snapshot (`QLEGS`). The request is non-blocking: the reply is processed in the normal event loop, so fills and status updates are never missed while the snapshot is outstanding. If the snapshot shows no legs, a different `quote_id`, or different leg order IDs than the bot is tracking, it clears its local state and reissues to converge.

11.8 Session state handling

The bot respects the exchange session lifecycle:

Session State	Bot Behaviour
PRE_OPEN	Wait — do not quote
OPENING_AUCTION	Cancel any live quote; wait
CONTINUOUS	Post and maintain a two-sided quote
CLOSING_AUCTION	Cancel any live quote; wait
CLOSED	Cancel any live quote; wait

A circuit-breaker halt is **not** a session.state value — it arrives on a separate, per-symbol circuit_breaker.halt.< topic (and clears on the matching circuit_breaker.resume.<SYMBOL> topic), independent of the session state. On a halt for one of its symbols, the bot cancels that symbol's live quote and pauses immediately; it resumes on the matching resume event.

When the session transitions to CONTINUOUS, the bot resumes quoting automatically.

11.9 Bootstrap: starting with an empty book

When the exchange starts fresh (no existing book or trades), the bot needs an initial reference price. It resolves one using this priority:

1. **QBOOT** — active quote from a previous session (restart recovery)
2. **Book mid / last trade** — the current book mid if another participant has posted orders, otherwise the most recent trade.executed price
3. **Bootstrap quote** — inactive quote prices from QBOOT
4. **Random range** — --initial_min to --initial_max (configurable)

If no source is available and no random range is configured, the bot exits with a clear error message.

```
# Bootstrap from random price range when the book is empty
pm-mm-bot --symbol AAPL --initial_min 95.00 --initial_max 105.00
```

11.10 Per-symbol settings

Settings fall into two tiers, and the tier decides where a setting may be given.

Gateway-wide settings describe the process itself. There is one gateway identity, one pair of sockets and one log stream, so these cannot vary per symbol:

Flag	Config-file location
--label	gateway.label

Flag	Config-file location
`--id-suffix`	`gateway.id_suffix`
`--gateway-id`	`gateway.gateway_id`
`--engine-pull`	`gateway.engine_pull`
`--engine-pub`	`gateway.engine_pub`
`--startup-session-timeout-sec`	`gateway.startup_session_timeout_sec`
`--shutdown-timeout-sec`	`gateway.shutdown_timeout_sec`
`--log-level`, `-v`, `-q`	`logging.level`
`--log-target`, `--log-file`, `--log-failover-timeout`	`logging.target`, `logging.file`, `logging.failover_timeout_sec`
`--config`	—

Per-symbol settings describe how one instrument is quoted, and may differ for every symbol:

`--strategy`, `--gap`, `--qty`, `--max-position`, `--drift-ticks`, `--tif`, `--reissue-delay-ms`, `--heartbeat-interval-sec`, `--bootstrap-timeout-sec`, `--cancel-timeout-sec`, `--qlegs-reconcile-interval-sec`, `--initial_min`, `--initial_ma`

11.10.1 Scoping flags to a symbol on the command line

`--symbol` may be repeated. Every per-symbol flag **after** a `--symbol` applies to that symbol until the next `--symbol`; anything **before** the first `--symbol` is the default for all of them:

```
pm-mm-bot \
  --label TECH --id-suffix 01 \
  --qty 500 --tif DAY --drift-ticks 3 \
  --symbol AAPL --gap 0.10 \
  --symbol MSFT --gap 0.20 --qty 200 \
  --symbol TSLA --gap 0.50 --qty 100 \
                --strategy inventory_skew --max-position 5000 \
                --initial_min 180 --initial_max 220
```

That resolves to gateway `MM_TECH_01` and:

Symbol	strategy	gap	qty	tif	drift_ticks	max_position
AAPL	symmetric	0.10	500	DAY	3	—
MSFT	symmetric	0.20	200	DAY	3	—
TSLA	inventory_skew	0.50	100	DAY	3	5000

Rules:

- A symbol that names no flags of its own inherits everything.
- `--symbols A,B,C` is shorthand for repeating `--symbol` with no per-symbol flags. It cannot be combined with `--symbol`.
- Naming the same symbol twice on one command line is a usage error.

- A gateway-wide flag typed after a `--symbol` still applies to the whole process (it is logged as a warning so nobody is misled) — this is what keeps every pre-existing single-symbol command line working unchanged.

Long command lines belong in a file

Anything beyond two or three symbols is much easier to read, review and version-control as a config file.

11.11 CLI reference

Argument	Tier	Default	Description
`--config PATH`	gateway	*unset*	YAML config file (see Config file)
`--symbol SYM`	*required*[1]	—	Instrument to make a market in; repeatable, and opens a scope for the per-symbol flags that follow it
`--symbols SYM1,SYM2,...`	*required*[1]	—	Comma-separated symbols sharing one set of settings — mutually exclusive with `--symbol`
`--all-symbols`	*required*[1]	off	Quote every symbol in the currently deployed configuration, all with the same settings; gateway ID defaults to `MM_ALL_<nn>`. Mutually exclusive with `--symbol`, `--symbols` and a config-file `symbols:` block
`--label NAME`	gateway	*derived*	Override the gateway-ID symbol segment (default: the single symbol, or every symbol joined with _)
`--id-suffix NN`	gateway	`01`	Running number for gateway ID (`MM_AAPL_01`)
`--gateway-id ID`	gateway	*derived*	Use this exact gateway ID (e.g. `MM01`) instead of `MM_<label>_<id-suffix>`; it must be a `MARKET_MAKER` gateway in the engine config
`--engine-pull ADDR`	gateway	`tcp://127.0.0.1:5555`	Engine PUSH/PULL address
`--engine-pub ADDR`	gateway	`tcp://127.0.0.1:5556`	Engine PUB address
`--startup-session-timeout-sec F`	gateway	`5.0`	Max wait to learn the session phase (asked for, or a `session.state`)
`--shutdown-timeout-sec F`	gateway	`2.0`	Max wait for cancel on SIGINT/SIGTERM
`--strategy NAME`	per-symbol	`symmetric`	Pricing strategy: `symmetric`, `inventory_skew` or `passive` (see Pricing strategies)
`--gap PRICE`	per-symbol	`0.10`	Total spread (bid at mid−gap/2, ask at mid+gap/2)

Argument	Tier	Default	Description
`--max-position N`	per-symbol	*unset*	Net position at which inventory skewing saturates — required with `--strategy inventory_skew`, rejected otherwise
`--qty N`	per-symbol	`500`	Quote size on each leg
`--drift-ticks N`	per-symbol	`3`	Reprice when mid moves by more than this many ticks (`passive`: when a side's target price does)
`--anchor-sim W`	per-symbol	`0` (off)	Quote around `pm-market-sim`'s true value: each model step closes `W` (0–1) of the gap, and the book stops setting the mid
`--retreat-ticks N`	per-symbol	`5`	`passive` only: how far beyond home each side may step back (see The control knobs)
`--behind-ticks N`	per-symbol	`1`	`passive` only: ticks behind other traders' best price on a covered side
`--min-cover-qty N`	per-symbol	`1`	`passive` only: others' quantity inside the band before a side counts as covered
`--fade-ticks N`	per-symbol	`2`	`passive` only: extra ticks a side steps back after it is filled (`0` = no fade)
`--fade-sec F`	per-symbol	`3.0`	`passive` only: how long a fade lasts (`0` = no fade)
`--reissue-delay-ms N`	per-symbol	`200`	Wait after fill before re-issuing
`--tif {DAY,GTC}`	per-symbol	`DAY`	Time-in-force for quote legs
`--heartbeat-interval-sec F`	per-symbol	`5.0`	Periodic live-quote check interval
`--bootstrap-timeout-sec F`	per-symbol	`1.0`	Max wait for QBOOT reply
`--cancel-timeout-sec F`	per-symbol	`1.0`	Max wait for cancel confirmation before forced replacement reissue
`--qlegs-reconcile-interval-sec F`	per-symbol	`15.0`	Periodic QLEGS reconciliation interval
`--initial_min PRICE`	per-symbol	*unset*	Lower bound for random bootstrap price
`--initial_max PRICE`	per-symbol	*unset*	Upper bound for random bootstrap price
`--log-level`	gateway	`WARNING`	Explicit level: `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG`

Argument	Tier	Default	Description
`-v`, `--verbose`	gateway	`false`	Increase verbosity (`-v` enables bot debug prints, `-vv` sets DEBUG)
`-q`, `--quiet`	gateway	`false`	Reduce output to warnings/errors

[1] At least one symbol is required, from `--symbol`, `--symbols`, or the config file's `symbols:` block. Giving both `--symbol` and `--symbols`, or neither, is a startup usage error.

`--all-symbols` reads the symbol list from the deployed compiled configuration (the one `pm-config-deploy` installs) when the bot starts, so the bot must run with the same data directory as the exchange. It exits with a usage error if nothing is deployed. Tuning flags and a config file's `gateway:`/`defaults:` blocks apply to every symbol; a config file used with `--all-symbols` may omit `symbols:` but must not name any.

11.12 Config file

`--config <path>` reads a YAML file. It is the recommended way to configure anything beyond a couple of symbols: the file is readable, reviewable, and can be committed alongside the `engine_config.yaml` it pairs with.

Worked examples live in `docs/examples/mm-bot/`.

11.12.1 File structure

A config file has four optional blocks and one required one:

```
version: 1        # schema version — always 1 today

gateway: {}       # gateway-wide settings (optional)
logging: {}       # logging settings (optional)
defaults: {}      # per-symbol settings applied to every symbol (optional)
symbols: {}       # REQUIRED — the symbols to quote, and their overrides
```

The split is the point: `gateway:` and `logging:` hold settings that *cannot* vary per symbol, `defaults:` holds the house settings for every symbol, and each block under `symbols:` overrides the house settings for one instrument.

11.12.2 A full example

```
# docs/examples/mm-bot/tech-desk.yaml
version: 1

# Gateway-wide settings. Nothing here can vary per symbol: there is one
# gateway identity and one pair of sockets for the whole process.
gateway:
  label: TECH                     # gateway id becomes MM_TECH_01
  id_suffix: "01"                 # quote it, or YAML reads 01 as the number 1
  engine_pull: tcp://127.0.0.1:5555
  engine_pub: tcp://127.0.0.1:5556
  startup_session_timeout_sec: 5.0
  shutdown_timeout_sec: 2.0

logging:
  level: INFO                     # CRITICAL | ERROR | WARNING | INFO | DEBUG
  target: server                  # server | stdout | file
  file: null                      # required when target: file
  failover_timeout_sec: 30

# Applied to every symbol below unless that symbol overrides the key.
defaults:
  strategy: symmetric
  gap: 0.10
  qty: 500
  drift_ticks: 3
  reissue_delay_ms: 200
  tif: DAY
  heartbeat_interval_sec: 5.0
  bootstrap_timeout_sec: 1.0
  cancel_timeout_sec: 1.0
  qlegs_reconcile_interval_sec: 15.0

# The symbol universe this bot quotes. Mapping order is the order symbols
# are started in.
symbols:
  # A liquid name, happy with every default.
  AAPL: {}

  # Wider and smaller than the house default.
  MSFT:
    gap: 0.20
    qty: 200

  # A volatile name: quote wide, in small size, skewing quotes to work
  # inventory back toward flat, and reprice less twitchily.
  TSLA:
    gap: 0.50
    qty: 100
    strategy: inventory_skew
    max_position: 5000
    drift_ticks: 5
    initial_min: 180
    initial_max: 220
```

```
pm-mm-bot --config docs/examples/mm-bot/tech-desk.yaml
```

That file produces gateway `MM_TECH_01` quoting `AAPL` on house defaults, `MSFT` twice as wide in less than half the size, and `TSLA` wide, small, inventory-skewing and less twitchy about repricing.

11.12.3 The smallest useful file

```
# docs/examples/mm-bot/single-symbol.yaml
# Equivalent to:  pm-mm-bot --symbol AAPL --gap 0.08 --qty 300
version: 1

defaults:
  gap: 0.08
  qty: 300

symbols:
  AAPL: {}
```

11.12.4 Shorthand when every symbol is the same

When no symbol needs its own block, `symbols:` may be a plain list instead of a mapping of empty blocks:

```
# docs/examples/mm-bot/uniform-desk.yaml
version: 1

gateway:
  label: CORE

defaults:
  gap: 0.10
  qty: 500
  tif: GTC

symbols: [AAPL, MSFT, TSLA]
```

11.12.5 `gateway:` fields

Field	Type	Default	Notes
`label`	string	derived from the symbols	Gateway ID becomes `MM_<label>_<id_suffix>`
`id_suffix`	string	`"01"`	**Quote it** — unquoted `01` is the number `1` to YAML, and is rejected
`gateway_id`	string	derived	Exact gateway ID; when set, `label` and `id_suffix` are ignored
`engine_pull`	string	`tcp://127.0.0.1:5555`	Engine PUSH/PULL address

Field	Type	Default	Notes
`engine_pub`	string	`tcp://127.0.0.1:5556`	Engine PUB address
`startup_session_timeout_sec`	number	`5.0`	Must be > 0
`shutdown_timeout_sec`	number	`2.0`	Must be > 0

11.12.6 `logging:` fields

Field	Type	Default	Notes
`level`	enum	`WARNING`	`CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG`
`target`	enum	`server`	`server`, `stdout`, `file`
`file`	string	`null`	Required when `target: file`
`failover_timeout_sec`	number	from log-client config	Grace window before falling back to a local file

11.12.7 `defaults:` and `symbols.<SYM>:` fields

The same keys are accepted in both places; a key in a symbol's own block wins for that symbol.

Field	Type	Default	Constraints
`strategy`	string	`symmetric`	`symmetric`, `inventory_skew` or `passive`
`gap`	number	`0.10`	> 0; must also satisfy the symbol's MM obligation (see Gap validation)
`qty`	integer	`500`	> 0
`max_position`	integer	`null`	> 0; **only** valid with `strategy: inventory_skew`, and **required** by it
`drift_ticks`	integer	`3`	> 0
`tif`	enum	`DAY`	`DAY` or `GTC`
`reissue_delay_ms`	integer	`200`	≥ 0
`heartbeat_interval_sec`	number	`5.0`	> 0
`bootstrap_timeout_sec`	number	`1.0`	> 0
`cancel_timeout_sec`	number	`1.0`	> 0

Field	Type	Default	Constraints
`qlegs_reconcile_interval_sec`	number	`15.0`	> 0
`initial_min`	number	`null`	Must be set together with `initial_max`
`initial_max`	number	`null`	Must be greater than `initial_min`
`retreat_ticks`	integer	`5`	0; `passive` only (ignored otherwise)
`behind_ticks`	integer	`1`	1; `passive` only
`min_cover_qty`	integer	`1`	1; `passive` only
`fade_ticks`	integer	`2`	0; `passive` only
`fade_sec`	number	`3.0`	0; `passive` only

Symbol keys are upper-cased on load, so `symbols: {aapl: {...}}` configures `AAPL`. Mapping order is preserved and is the order symbols start in.

11.12.8 Precedence

When the same per-symbol setting is given in more than one place, the first of these that supplies it wins:

Rank	Source	Example
1	The symbol's own CLI scope	`--symbol MSFT --gap 0.20`
2	The symbol's block in the config file	`symbols.MSFT.gap`
3	The global CLI scope	`--gap 0.20` before the first `--symbol`
4	The config file's `defaults:` block	`defaults.gap`
5	The built-in default	`0.10`

Gateway-wide settings use the simpler ladder: the CLI flag, then the file's `gateway:`/`logging:` block, then the built-in default.

A per-symbol file value beats a gateway-wide flag

Rank 2 sits above rank 3 deliberately. If `symbols.TSLA.gap: 0.50` is written into a reviewed config file, a throwaway `--gap 0.10` on the command line widens the symbols that were *not* individually tuned — it does not silently un-tune TSLA. To override one symbol from the command line, name it: `--symbol TSLA --gap 0.10`.

A gap chosen at *any* of ranks 1–4 counts as an explicit choice, so the bot will not replace it with the MM-obligation default described in Gap validation. Only a gap nobody set is auto-derived.

11.12.9 Errors

The file is validated up front; a bad file is a fast, explicit startup failure rather than a bot that runs with settings you did not intend.

Mistake	What you see
Typo in a key	`unknown key 'gapp' under symbols.AAPL (did you mean 'gap'?)`
Gateway-wide key in `defaults:` or a symbol block	`'engine_pull' is a gateway-wide setting; move it under 'gateway:'`
Per-symbol key in `gateway:`	`'gap' is a per-symbol setting; move it under 'defaults:' or into a symbol's own block`
Logging key in the wrong place	`'log_level' is a logging setting; move it under 'logging:'`
Unquoted `id_suffix`	`gateway.id_suffix must be a string — quote it (e.g. "01")`
A future schema version	`unsupported version 2 (this build understands version 1)`
An out-of-range value	`[MSFT] qty must be positive (got -1)`
`inventory_skew` without a cap	`[TSLA] max_position is required when strategy inventory_skew is selected`

11.12.10 Legacy flat files

The original flat format — one mapping of CLI flag names, all of them gateway-wide — still loads unchanged:

```
# mm_aapl.yaml (legacy format)
symbol: AAPL
id_suffix: "01"
strategy: symmetric
gap: 0.08
qty: 300
tif: GTC
drift_ticks: 4
```

It is recognised by shape (no `version:`, and `symbols:` given as a string or list rather than a mapping) and behaves exactly as it did: every value applies to every symbol. Prefer `version: 1` for new files — it is the only format that can express per-symbol settings.

11.12.11 Pricing strategies

`--strategy` (or the file's `strategy:` key) selects which pricing logic the bot uses to compute bid/ask from the tracked mid-price. Three strategies ship today:

- **`symmetric`** (the default) — quote symmetrically around mid at a fixed `--gap`, described in Pricing logic above.
- **`inventory_skew`** — quote asymmetrically to lean the book toward flattening whatever position the bot has accumulated, described in Inventory skewing below.
- **`passive`** — act as a backstop: quote at home when nobody else is there, step behind other traders when they are, and back off briefly after a fill. Described in The passive strategy below.

The bot fails fast at startup if `--strategy` names anything else. The selection point exists so a future strategy

(e.g. one that widens the gap with volatility) can be added as a new pricing module without changing the bot's state machine, ZMQ handling, or CLI plumbing.

11.12.12 Inventory skewing

A market maker that always quotes symmetrically around mid has no opinion about the position it is carrying. If the bot has been repeatedly lifted on its ask (selling to takers), it accumulates a growing short position while continuing to offer the exact same bid and ask it always did — nothing about its quoting nudges the market back toward flattening that position. In a real market this matters because holding inventory is risk: the longer the bot stays short (or long), the more exposed it is to an adverse price move, and the point of market-making in the first place is to earn the spread, not to take a directional bet.

`--strategy inventory_skew` addresses this by tracking the bot's own net position per symbol (derived locally from its own fills — it does not ask the engine) and shifting where it places its bid and ask relative to mid:

- **Long** (net_position > 0, the bot has bought more than it's sold): the effective mid shifts *down*. Both the bid and the ask come down — the ask becomes more attractive to lift (encouraging someone to buy from the bot, reducing the long) and the bid becomes less attractive to hit (discouraging the bot from buying still more).
- **Short** (net_position < 0): the shift is the mirror image — both quotes move *up*, encouraging the bot to buy back what it's short and discouraging it from selling further.
- **Flat** (net_position $== 0$): identical to `symmetric` — no shift at all.

The shift scales linearly with how large the position is relative to `--max-position` (required for this strategy — there is no meaningful "unbounded" skew to normalize against):

```
fraction = clamp(net_position / max_position, -1.0, +1.0)
skew     = -fraction × (gap / 2)
effective_mid = tracked_mid + skew
```

The bid and ask are then computed from `effective_mid` exactly as `symmetric` computes them from the tracked mid (same tick-rounding, same 2-tick minimum spread) — inventory skewing only changes *where* the quote is centred, never the rules for how it's rounded or how wide it can get.

Worked example. AAPL, `--gap 0.10`, `--max-position 1000`, tracked mid = `150.00`. At `net_position = +500` (half of max_position, long): `fraction = 0.5`, `skew = -0.5 × 0.05 = -0.025`, `effective_mid = 149.975`. The bid and ask (still `±0.05` from that effective mid, then tick-rounded) come out at `149.92` / `150.02` — both about three cents lower than the flat-position quote of `149.95` / `150.05`, making the ask cheaper to lift and the bid less attractive to hit. (The rounded values land three cents, not the two-and-a-half the raw `149.975 ± 0.05` arithmetic suggests, because `149.975 - 0.05` evaluates to `149.92499999999998` in floating point and rounds down a tick.)

What happens when `--max-position` is reached? The bot does **not** stop quoting. `fraction` is clamped to `±1.0`, so once `|net_position| >= max_position` the skew simply stays pinned at its maximum (the full `±gap/2` shift) — the spread stays maximally lopsided in the direction that favors flattening, and further fills past the cap do not skew it any further, but the bot keeps posting a live two-sided quote throughout. This is a deliberate choice: pulling quotes entirely at the position cap would leave the symbol with no MM liquidity at exactly the moment the bot most needs fills to flatten, which defeats the purpose. If you want the bot to stop trading altogether at some limit, that is a different, not-yet-built feature (a hard risk cutoff) — `--max-position` here only ever affects how the quote is skewed, never whether one is sent.

`inventory_skew` accepts `--gap`, `--drift-ticks`, and all the other `symmetric` flags unchanged — only the price-

computation step differs.

11.12.13 The passive strategy

`symmetric` and `inventory_skew` both centre their quote on the mid of the *whole* book — including the bot's own quote — and always quote at `mid ± gap/2`. On a quiet book that makes the bot the best bid and the best offer almost all the time: it re-quotes 200 ms after every fill, so whenever a trader wants to trade immediately the bot is the one that gets hit, and a trader who rests a limit order at the bot's price rarely gets filled first.

Real market makers are not obliged to be the *best* price. Their obligation is to be *present*: quote both sides, within a maximum spread, for most of the session. Being at the top of the book is a business choice. `--strategy passive` models that: the bot acts as a **backstop**. It makes the market when nobody else does, and steps back behind other traders when they are already providing liquidity.

The strategy combines two behaviours:

- **Step behind** — when other traders already quote near the bot's price, the bot moves its quote *behind* theirs, so their orders trade first.
- **Fade** — after one of its legs is filled, the bot quotes that side a little further out for a few seconds, instead of immediately re-offering the same price.

```
pm-mm-bot --symbol AAPL --strategy passive
```

With no other flags every knob below takes its default.

Vocabulary

The rules are easiest to follow with four terms. The examples use a `0.01` tick, `--gap 0.10` and the defaults (`--retreat-ticks 5 --behind-ticks 1 --min-cover-qty 1 --fade-ticks 2 --fade-sec 3`).

Term	Meaning	Example (mid `100.00`)
Others' book	The order book with the bot's own two legs removed	—
Home price	Where `symmetric` would quote: `mid ± gap/2`, tick-rounded. The *tightest* the bot ever quotes	bid `99.95`, ask `100.05`
Band	From the home price out to `retreat_ticks` further away. The bot never quotes outside it	bid `99.90`–`99.95`, ask `100.05`–`100.10`
Covered side	A side where other traders show at least `min_cover_qty` in total at prices inside that side's band (or better)	—

```
        bid band              ask band
    |<--- retreat --->|   |<--- retreat --->|
 99.90             99.95 | 100.05            100.10
 floor              home | home              ceiling
                       mid
                    100.00
```

Step 1 — the mid comes from other traders only

Before pricing anything the bot removes its own legs from every book snapshot. It knows where they rest (it sent them) and how much of each is left (it sees its own fills). The mid is then computed from what remains, with the usual rule (both sides → average; one side → that side; nothing → keep the previous mid).

This is what stops the bot chasing itself. Under `symmetric`, the bot's own quote *is* part of the book it prices from; under `passive`, a book containing only the bot's quote is an empty book, and the mid stays where it was.

How own-leg removal works

A snapshot level is reduced by the bot's leg only if it is at the bot's price and holds at least the leg's remaining quantity. If that level holds exactly one order, the level is dropped. A snapshot taken just before the leg reached the book (or just after it left) is therefore passed through unchanged instead of having another trader's quantity subtracted. The next snapshot corrects any leftover difference.

Step 2 — place each side

Each side is priced independently. For the bid:

```
home   = mid − gap/2                          (tick-rounded)
floor  = home − retreat_ticks
covered = (others' bid qty at prices ≥ floor) ≥ min_cover_qty

if covered:  bid = clamp(others' best bid − behind_ticks,  floor,  home)
else:        bid = home
```

The ask is the mirror image: `ceiling = home + retreat_ticks`, count others' offers at prices ≤ ceiling, and quote `behind_ticks` *above* their best offer, clamped to `[home, ceiling]`.

Three consequences follow from the clamp:

- The bot **never quotes tighter than home**. If other traders are already tighter than the bot's gap, the bot simply stays at home, which is already behind them.
- The bot **never quotes outside the band**. If other traders sit at the edge of the band, the bot joins them at the edge. It arrives later, so under price-time priority it is still behind them in the queue.
- Liquidity **outside the band doesn't count**. A resting order far from the mid doesn't make the bot step back; the bot is still the market between the mid and that order.

Step 3 — fade after a fill

When a leg is filled, that side is pushed `fade_ticks` further out, still clamped to the band, for `fade_sec` seconds. The other side is unchanged. Another fill on the same side restarts the timer (fades don't stack). When the timer runs out the bot re-quotes that side back at its normal price.

Fading composes with the reissue delay rather than replacing it. After a fill the engine inactivates the quote. The bot waits `--reissue-delay-ms` as always, then sends a new quote that already includes the fade.

Worked examples

All with mid `100.00` unless stated; qty is the other trader's order size.

Others' book	Mid	Bid	Ask	Why
empty	100.00 (kept)	99.95	100.05	Nobody else → the bot is the market, at home
bid 200 @ 99.97, ask 200 @ 100.03	100.00	99.95	100.05	Both sides covered, but others are tighter than home → stay at home, already behind them
bid 200 @ 99.93, ask 200 @ 100.07	100.00	99.92	100.08	Covered inside the band → one tick behind them
bid 200 @ 99.91, ask 200 @ 100.09	100.00	99.90	100.10	Behind them would be 99.90/100.10, exactly the band edge
bid 200 @ 99.85, ask 200 @ 100.15	100.00	99.95	100.05	Their orders are outside the band → not covered → the bot quotes at home, in front of them
bid 200 @ 99.97, no asks	99.97	99.92	100.02	One-sided book → mid is that bid. Bid covered (home 99.92 is already behind), ask not covered → at home
empty, bot's bid just filled	100.00	99.93	100.05	Bid faded 2 ticks for 3 s
bid 200 @ 99.93, ask 200 @ 100.07, bot's bid just filled	100.00	99.90	100.08	Behind (99.92) minus fade (2) = 99.90, the floor

What a trader sees

With `passive` the bot's quote is the price a trader can always rely on, but not necessarily the best one. A trader who posts a buy inside the bot's spread becomes the best bid and gets filled first. If the trader posts *at* the bot's bid, the bot notices on the next book update and steps back one tick. The trader then has the top of the book to themselves.

When the bot re-quotes

Moving a quote means cancelling and re-issuing it, so the bot doesn't react to every change in the book. After each quote it remembers, per side, the price it sent, whether the side was covered, and whether it was faded. It re-quotes when any of these hold for either side:

Condition	Reaction	Reason
The side became covered, or stopped being covered	Immediately	The bot must step back behind a newcomer, or step forward when the other trader leaves
A covered side must move *further back* (by any amount)	Immediately	The bot never stays level with or ahead of the traders it is yielding to
A fade started or ended	Immediately (a fade ending is checked on the bot's timer, even with no book activity)	The fade is time-driven, not book-driven
Any other move of the target price by **more than** `--drift-ticks`	Re-quote	Same laziness as `symmetric`; a covered side creeping *closer* to the market, or a small mid move, waits until it's worth a cancel/replace

So for `passive`, `--drift-ticks` measures how far a *side's target price* has moved, not how far the mid has moved.

Interaction with the MM spread obligation

If the symbol has an `mm_max_spread_ticks` obligation (see Gap validation), the engine rejects any quote wider than it. The bot therefore narrows the band until even the widest quote it could send (both sides fully retreated, or faded to the band edge) still satisfies the obligation:

```
effective retreat = min(retreat_ticks, (mm_max_spread_ticks − home spread in ticks) // 2)
```

Example: obligation `14` ticks, home spread `10` ticks → at most `2` ticks of retreat per side, whatever `--retreat-ticks` says. The bot logs one line at startup when this narrowing applies:

```
[AAPL] retreat_ticks 5 narrowed to 2 by mm_max_spread_ticks=14
```

When no gap is set, the gap defaults to half the obligation, which leaves a quarter of the obligation for retreat on each side. With a `20`-tick obligation, for example, the home spread is `10` ticks and each side can retreat `5`, which is exactly the default `--retreat-ticks 5`.

The control knobs

All are per-symbol (CLI scope, `defaults:` or `symbols.<SYM>:`), have the defaults below, and are **ignored by every strategy except `passive`**.

Knob	CLI flag	Default	Range	Effect	Turn it up to…	Turn it down to…
`gap`	`--gap`	`0.10`	> 0	Width of the home quote — the tightest the bot ever quotes	Quote less aggressively when alone	Make a tighter market when alone
`retreat_ticks`	`--retreat-ticks`	`5`	0	How far out each side may step back. `0` disables stepping behind (and fading)	Yield to traders further from the mid	Stay close to home; `0` = never yield
`behind_ticks`	`--behind-ticks`	`1`	1	How many ticks behind others' best price a covered side sits	Leave a visible gap behind other traders	Sit right behind them
`min_cover_qty`	`--min-cover-qty`	`1`	1	Total quantity others must show inside the band before the bot yields	Keep a small order from pushing the bot away	Yield to any order at all
`fade_ticks`	`--fade-ticks`	`2`	0	How far a side steps back after it is filled. `0` disables fading	Back off harder after being hit	Refill closer to the original price
`fade_sec`	`--fade-sec`	`3.0`	0	How long a fade lasts. `0` disables fading	Stay backed off longer, giving others time to post	Return sooner
`drift_ticks`	`--drift-ticks`	`3`	> 0	Target-price move (per side) that triggers a lazy re-quote	Fewer cancel/replace cycles	Track the book more closely

Knob	CLI flag	Default	Range	Effect	Turn it up to…	Turn it down to…
`reissue_delay_ms`	`--reissue-delay-ms`	200	0	Pause after a fill before the new quote goes out	Leave a longer gap with no MM quote at all after each fill	Refill faster

Fade vs. reissue delay

Both slow the bot down after a fill, but differently. `reissue_delay_ms` is a period with **no MM quote at all**, and on a real exchange that counts against a market maker's presence obligation. Fading keeps a quote in the book, just further out. To give other traders room after a fill, prefer a longer `fade_sec` over a longer `reissue_delay_ms`.

Recipes

Yield to anyone, but only a little:

```
pm-mm-bot --symbol AAPL --strategy passive --retreat-ticks 2
```

Only yield to real size (at least one round lot inside the band):

```
pm-mm-bot --symbol AAPL --strategy passive --min-cover-qty 100
```

Pure step-behind, no fading:

```
pm-mm-bot --symbol AAPL --strategy passive --fade-ticks 0
```

Pure fading, never step behind others. The band still has to leave room for the fade:

```
pm-mm-bot --symbol AAPL --strategy passive --min-cover-qty 1000000000 --retreat-ticks 3 --fade-ticks 3
```

The same in a config file, with one symbol tuned differently:

```
version: 1
gateway:
  id_suffix: "01"
defaults:
  strategy: passive
  gap: 0.10
  retreat_ticks: 5
  behind_ticks: 1
  min_cover_qty: 1
  fade_ticks: 2
  fade_sec: 3.0
symbols:
  AAPL: {}
  TSLA:
    gap: 0.30
    retreat_ticks: 10
    min_cover_qty: 200
    fade_sec: 5.0
```

Limitations

- The bot has no fair value of its own; its reference is the others' mid. A lone order far from the market on an otherwise empty side moves the mid to that order, exactly as for `symmetric`.
- Covered is judged from aggregated price levels, so the bot cannot tell one trader with 500 from five traders with 100.
- `passive` doesn't skew for inventory; that is `inventory_skew`'s job, and the two can't be combined today.

11.12.14 Querying a bot's position

Whether or not `inventory_skew` is in use, an operator can ask any running gateway — including an MM bot — what it is currently holding, from `pm-alf-console`:

```
POS|GW=MM_AAPL_01
```

This sends `system.position_request` for that `gateway_id` and prints whatever the engine's own per-gateway position ledger reports back on `system.position_snapshot.MM_AAPL_01`: net quantity and average cost, per symbol, for every symbol the bot has a non-zero position in. `POS` with no `GW=` is unchanged — it still shows the console's own local fills and P&L.

This does not require any special support from the MM bot itself: the engine tracks a signed net position and volume-weighted average cost for every gateway from its own fill records, and answers `system.position_request` for any `gateway_id`, not just the requester's own — a console (or another script) can query about a bot the same way it would query its own position. The `--strategy inventory_skew` bot's local position tracking (used for skewing, described above) is a separate, parallel bookkeeping of the same fills — the two should always agree, since both are derived from the same fill stream, but only the engine's ledger is reachable remotely via `POS|GW=`.

The same query is available over REST for an admin-role caller, via `GET /api/v1/admin/gateways/MM_AAPL_01/posi` — useful for a dashboard or a script that wants to watch a bot's position without an interactive console session. It reads the same `system.position_request`/`system.position_snapshot` pair `POS|GW=` uses, so both surfaces always agree.

A fourth surface, `pm-admin-cli position --gw MM_AAPL_01`, adds the bot's **live quote** to the same net-position figures — bid, ask, spread, and mid derived from its currently resting quote order, alongside `net_qty`/`avg_cost` — in one command, with a `--format json` option for scripting. Handy for confirming that inventory skewing is actually doing something: watch `spread`/`mid` shift as `net_qty` moves away from zero.

11.13 Engine configuration

11.13.1 Gateway registration

Each bot instance — whether it quotes one symbol or several — must be pre-registered as a single gateway entry in `engine_config.yaml`:

```
participants:
  - id: MM_AAPL_01
    description: "AAPL market-maker bot instance 1"
    role: MARKET_MAKER
    disconnect_behaviour: CANCEL_QUOTES_ONLY
    quote_refresh_policy: INACTIVATE_ON_ANY_FILL
    enforce_mm_obligation: true
    mm_max_spread_ticks: 10
    mm_min_qty: 100
    smp_action: CANCEL_RESTING

  - id: MM_AAPL_02
    description: "AAPL market-maker bot instance 2"
    role: MARKET_MAKER
    disconnect_behaviour: CANCEL_QUOTES_ONLY
    quote_refresh_policy: INACTIVATE_ON_ANY_FILL
```

A `pm-mm-bot --symbols AAPL,MSFT --label TECH` process registers as *one* gateway entry the same way — nothing in `participants:` names which symbols a `MARKET_MAKER` gateway quotes, since that's the bot's own `--symbol`/ `--symbols` choice, not an engine-config concept:

```
participants:
  - id: MM_TECH_01
    description: "AAPL+MSFT market-maker bot"
    role: MARKET_MAKER
    disconnect_behaviour: CANCEL_QUOTES_ONLY
    quote_refresh_policy: INACTIVATE_ON_ANY_FILL
    enforce_mm_obligation: true
    mm_max_spread_ticks: 10
    mm_min_qty: 100
```

One easy-to-miss consequence: the per-symbol `market_maker_quotes` seed (needed to bootstrap a fresh exchange — see Bootstrap) is required on **every** symbol once **any** `MARKET_MAKER` gateway exists in the config, regardless of which gateway is meant to quote which symbol (`pm-cverifier`'s check `M001` enforces this). So `MM_TECH_01`'s config needs a seed under both `symbols.AAPL.market_maker_quotes` and `symbols.MSFT.market_maker_q` each referencing `MM_TECH_01` — `pm-config-gen --seed-mm-mid-range` generates these automatically for every symbol, so this is only a concern when writing `engine_config.yaml` by hand.

11.13.2 Recommended settings

- **`disconnect_behaviour: CANCEL_QUOTES_ONLY`** — ensures stale quotes are removed if the bot crashes or restarts
- **`quote_refresh_policy: INACTIVATE_ON_ANY_FILL`** — the engine cancels the remaining leg when either side fills, triggering an immediate reissue
- **`smp_action`** — self-match-prevention default for this gateway (`NONE` if unset). Since `mm_bot` only ever submits `QUOTE`s (no plain `NEW`/combo orders), this is the *only* SMP control it has — quote legs have no per-request `SMP=` field of their own, so a bid/ask leg sweeping into a resting order from the *same* gateway id (e.g. a stale leg left over from a prior quote) is always handled per this gateway-level setting instead of self-trading. See Configuration Spec §5.2 for the full `SmpAction` value list and how this default also applies to `NEW`/combo orders from other gateway roles when they omit `SMP=`. See Risk Controls — Self-Match Prevention for the full conceptual explanation, including a worked example of exactly this stale-quote-leg scenario

11.13.3 Gap validation

The bot enforces pricing validity at startup, per symbol (each symbol has its own `tick_size` and, potentially, its own `mm_max_spread_ticks`):

- `gap >= 2 * tick_size` (via pricer validation)
- if `mm_max_spread_ticks` is available in that symbol's metadata, the bot validates `gap <= mm_max_spread_ticks * tick_size`

Defaulting rules:

- An explicitly supplied `--gap` (in either the `--gap 0.10` or `--gap=0.10` form) is always respected for every symbol — it is only validated against each symbol's obligation, never overridden.
- If `--gap` is not provided and `mm_max_spread_ticks` is available for a symbol, that symbol's gap defaults to half its max spread: `(mm_max_spread_ticks / 2) * tick_size`. On a `--symbols` bot this can differ per symbol — `AAPL` and `MSFT` need not end up with the same effective gap.
- If no MM spread metadata is available for a symbol, that symbol uses the standard `0.10` default.

A symbol whose validated gap violates its own obligation is excluded from quoting rather than failing the whole process — see Per-symbol failure isolation above.

11.14 Usage examples

11.14.1 Single symbol, default settings

```
pm-mm-bot --symbol AAPL
```

11.14.2 One process quoting several symbols, same settings

```
pm-mm-bot --symbols AAPL,MSFT,TSLA --label TECH
```

11.14.3 One process, different settings per symbol

```
pm-mm-bot --label TECH --qty 500 \
  --symbol AAPL --gap 0.10 \
  --symbol MSFT --gap 0.20 --qty 200 \
  --symbol TSLA --gap 0.50 --qty 100 --strategy inventory_skew --max-position 5000
```

11.14.4 The same desk, from a config file

```
pm-mm-bot --config docs/examples/mm-bot/tech-desk.yaml
```

11.14.5 Two competing MMs on the same symbol

```
pm-mm-bot --symbol AAPL --gap 0.08 --qty 500 &
pm-mm-bot --symbol AAPL --gap 0.12 --qty 300 --id-suffix 02 &
```

11.14.6 Faster repricing for volatile sessions

```
pm-mm-bot --symbol MSFT --gap 0.20 --drift-ticks 1 --reissue-delay-ms 100
```

11.14.7 Fresh exchange with no book data

```
pm-mm-bot --symbol AAPL --initial_min 95.00 --initial_max 105.00
```

11.14.8 Verbose mode for troubleshooting

```
pm-mm-bot --symbol AAPL --gap 0.10 --qty 500 -v
```

11.14.9 Config file, with one value overridden on the CLI

```
# Widens every symbol that has no gap of its own in the file
pm-mm-bot --config mm_tech.yaml --gap 0.12

# Widens only MSFT, whatever the file says for it
pm-mm-bot --config mm_tech.yaml --symbol MSFT --gap 0.12
```

11.14.10 A market maker that lets other traders go first

```
pm-mm-bot --symbol AAPL --strategy passive --gap 0.10 --retreat-ticks 5 --fade-sec 5
```

11.14.11 Inventory skewing with a position cap

```
pm-mm-bot --symbol AAPL --strategy inventory_skew --max-position 1000 --gap 0.10
```

11.14.12 Querying that bot's position from another console

```
POS|GW=MM_AAPL_01
```

11.15 Understanding bot output

Every log line has the standard EduMatcher format (`<timestamp> <LEVEL> <logger-name> - <message>`), and every message the bot itself emits is further prefixed with its own gateway ID in brackets, e.g. `[MM_AAPL_01] <message>`. At true default verbosity (no `-v`/`-q`/`--log-level` flag at all) the bot's own lifecycle messages are logged at `INFO`, but the default logging level is `WARNING` — so **none of them are visible** unless something fails outright (those paths log at `ERROR`). Pass `-v` to see the bot's milestones and problems (routine per-quote activity needs `-vv` or the dedicated `--verbose` sections below):

```
2026-09-20 09:30:00,001 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] starting: gateway=MM_AAPL_01 symbols=AAPL
2026-09-20 09:30:00,002 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] [AAPL] strategy=symmetric gap=0.1 qty=500 tif=DA
2026-09-20 09:30:01,002 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] authenticated
2026-09-20 09:30:01,003 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] bootstrap from random range: 150.00
2026-09-20 09:30:20,004 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] [AAPL] circuit breaker HALT
2026-09-20 09:30:45,005 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] [AAPL] circuit breaker RESUME
2026-09-20 09:35:00,006 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] shutdown complete
```

The `starting:` line names the gateway and every symbol the process covers, followed by **one line per symbol** reporting that symbol's resolved settings. This is the confirmation that a long command line or a config file resolved the way you intended — read it first when a bot is not quoting the way you expected. Every per-symbol log line thereafter carries an additional `[SYMBOL]` tag ahead of the message:

```
2026-09-20 09:30:00,001 INFO edumatcher.mm_bot.bot - [MM_TECH_01] starting: gateway=MM_TECH_01 symbols=AAPL,MSFT
2026-09-20 09:30:00,002 INFO edumatcher.mm_bot.bot - [MM_TECH_01] [AAPL] strategy=symmetric gap=0.1 qty=500 tif=D
2026-09-20 09:30:00,003 INFO edumatcher.mm_bot.bot - [MM_TECH_01] [MSFT] strategy=symmetric gap=0.2 qty=200 tif=D
2026-09-20 09:30:01,002 INFO edumatcher.mm_bot.bot - [MM_TECH_01] authenticated
2026-09-20 09:30:01,003 INFO edumatcher.mm_bot.bot - [MM_TECH_01] [AAPL] bootstrap from random range: 150.00
2026-09-20 09:30:01,004 INFO edumatcher.mm_bot.bot - [MM_TECH_01] [MSFT] bootstrap from random range: 310.00
2026-09-20 09:30:14,005 INFO edumatcher.mm_bot.bot - [MM_TECH_01] [AAPL] startup failed: --gap exceeds mm_max_spr
2026-09-20 09:30:14,006 INFO edumatcher.mm_bot.bot - [MM_TECH_01] [AAPL] excluded from quoting: --gap exceeds mm_
2026-09-20 09:30:14,007 INFO edumatcher.mm_bot.bot - [MM_TECH_01] running symbols=['MSFT'] session=CONTINUOUS
2026-09-20 09:30:20,008 INFO edumatcher.mm_bot.bot - [MM_TECH_01] [MSFT] circuit breaker HALT
2026-09-20 09:30:45,009 INFO edumatcher.mm_bot.bot - [MM_TECH_01] [MSFT] circuit breaker RESUME
2026-09-20 09:35:00,010 INFO edumatcher.mm_bot.bot - [MM_TECH_01] shutdown complete
```

Note that a symbol excluded during startup (as `AAPL` is here) never reaches the point of sending a `QUOTE` to the engine, so there is no engine-side `quote REJECTED` for it — the `startup failed:`/`excluded from quoting:` pair above is the complete picture for that symbol.

Here `AAPL` failed its gap-vs-obligation check at startup (see Per-symbol failure isolation) and was excluded, while `MSFT` continued on its own — the `running symbols=[...]` line always reports exactly which symbols made it into the main quoting loop.

With `--verbose` (`-v`), the bot additionally logs routine quoting activity — symbol/session updates, every quote sent, every fill, and every repricing decision:

```
2026-09-20 09:30:01,001 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] symbols received: ['AAPL', 'MSFT', 'TSLA']
2026-09-20 09:30:01,002 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] [AAPL] reference from book/trade: 150.00
2026-09-20 09:30:01,003 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] [AAPL] QUOTE sent bid=149.95 ask=150.05
2026-09-20 09:30:01,004 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] [AAPL] quote ACK id=q-001
2026-09-20 09:30:14,005 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] [AAPL] fill: ASK 200@150.05
2026-09-20 09:30:15,006 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] [AAPL] QUOTE sent bid=149.95 ask=150.05
2026-09-20 09:31:02,007 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] [AAPL] drift detected — repricing
2026-09-20 09:31:02,008 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] [AAPL] state: QUOTING -> REPRICING
2026-09-20 09:31:02,009 INFO edumatcher.mm_bot.bot - [MM_AAPL_01] [AAPL] QUOTE sent bid=149.99 ask=150.09
```

(`symbols received:` covers the whole gateway and carries no `[SYMBOL]` tag; every other line above is per-symbol and always carries one, even when the bot is only quoting a single symbol.)

`-v` also raises the underlying log level and enables the bot's own debug prints together — it is not just a "print more" switch, it also turns on low-level flow tracing (e.g. `book mid=...`, `session: OLD -> NEW`, and `state: OLD -> NEW` transitions of the bot's own state machine).

11.16 Troubleshooting

Symptom	Cause	Fix
`auth rejected`	Gateway ID not in `engine_config.yaml`	Add the `MM_<SYM>_<nn>` (or `MM_<LABEL>_<nn>`) entry with `role: MARKET_MAKER`

Symptom	Cause	Fix
`invalid config file: ... unknown key(s)`	A `--config` file has a typo'd or unsupported key	Check the key against Config file — long flag name, dashes as underscores
`no symbols configured: use --symbol, --symbols, or a config file with a 'symbols:' block`	Neither `--symbol` nor `--symbols` nor the config file's `symbols:` block was given	Add one of the two
`--symbol and --symbols are mutually exclusive`	Both `--symbol` and `--symbols` ended up set, from any combination of CLI and `--config`	Use only one
`startup failed: no reference price available (no book, no trade, no bootstrap, no random range)`	Empty book + no `--initial_min`/`--initial_max`, for a symbol with no other symbol left quoting	Add bootstrap range flags
`auth rejected: Gateway already connected: <ID>`	A bot with that gateway ID is still logged on, or one died without logging off	Stop the other bot; after a crash, wait: the bot heartbeats the engine once a minute, and the engine frees the ID after three missed beats
`startup failed: no session.state`	Engine not running or scheduler not started	Start the engine and scheduler
`startup failed: no symbol survived startup checks`	Every symbol failed startup (see Per-symbol failure isolation)	Fix whichever per-symbol cause each excluded-symbol log line names
`[SYM] excluded from quoting: ...`	One symbol (not all) failed a startup check on a `--symbols` bot	Expected if intentional (e.g. testing failure isolation); otherwise fix that symbol's cause and restart
`quote REJECTED`	Gap or qty violates MM obligation policy	Reduce `--gap` / increase `--qty` or adjust gateway MM settings
Bot quotes but prices look wrong	Tick size mismatch	Check symbol `tick_size` in engine config
`passive` bot never steps back behind other traders	Their orders are outside the band, below `--min-cover-qty`, or the band was narrowed to `0` by the MM spread obligation (look for `retreat_ticks ... narrowed to 0` at startup)	Raise `--retreat-ticks`, lower `--min-cover-qty`, or narrow `--gap` so the obligation leaves room
`passive` bot still gets hit first after a trader joins its price	The bot was at that price first, so it has time priority until its cancel/replace goes through	Expected briefly — the bot steps back as soon as it processes the book update

Symptom	Cause	Fix
`[SYM] behind_ticks must be >= 1` (or similar)	A `passive` knob is out of range	See the range column in The control knobs
Bot stops quoting for a while, then resumes on its own	An engine reply (`quote.ack`/`quote.status`) was dropped	Expected self-healing; the heartbeat and QLEGS reconciliation recover automatically. Lower `--heartbeat-interval-sec` for faster recovery

---

11.17 See also

In this book:

- Market Making — the `QUOTE` command, the MM obligations framework, and Market-Maker Protection (MMP)
- AI Traders — the `pm-ai-trader` and `pm-ai-swarm` processes

Operator's Guide:

- Configuration — engine and gateway configuration
- Risk Controls — the kill switch

Reference Manual:

- Processes — architecture overview of all EduMatcher processes

CHAPTER 12

AI Traders

Learning objectives

After reading this page you will understand:

- What `pm-ai-trader` and `pm-ai-swarm` are and how they reach the exchange
- What a *preset* is, and which built-in presets there are
- How to run one AI trader, a swarm of them, and your own preset
- Which traders need the market model (`pm-market-sim`) and why

Prerequisites: Running the Exchange, ALF Gateway — AI traders log on through `pm-alf-gwy` like any ALF client, so their IDs must be participants in the deployed configuration.

12.1 What are the AI trader processes?

Process	Purpose
`pm-ai-trader`	One AI trader, logged on as one participant
`pm-ai-swarm`	Many AI traders — tens to hundreds — in a few worker processes, each trader logged on as its own participant

An AI trader is a participant like a student desk. It logs on to `pm-alf-gwy` with its own ID (`AI001`, `AI002`, …, role `TRADER`), sends its orders over that ALF session, and reads market data from the engine's public feed. Everything it does is visible to everyone else exactly as a person's orders are: in the books, the trades, the audit trail. When its session ends, the participant's `disconnect_behaviour` applies (`CANCEL_ALL` for the example AI participants).

The `s150-*` examples register `AI001`–`AI020`, the `s300-load` example `AI001`–`AI500` and the `s300-sim` example `AI001`–`AI100`; all ship a `swarm.yaml` and a `market_sim.yaml`. `s300-sim` is the long-running simulated market of the AI Traders and Market Simulation book.

12.2 Presets

What a trader does is set by its *preset*, four parts:

Part	Decides	Choices
Strategy	*What* it wants to trade, which way, how urgently	`noise`, `trend`, `reversion`, `value`, `news`
Execution	*How* a wish becomes orders	`passive`, `marketable`, `sweep`, `iceberg`, `twap`
Tempo	How often it decides, and how big its orders are	`many-small`, `aggressive`, `cautious`, `few-large`, or your own
Risk	Position limit, live orders per symbol, order age, protective orders	per preset

How the AI Traders Decide explains each in detail. The built-in presets (`pm-ai-trader --list-presets`):

Preset	Strategy	Execution	Tempo	Protection	Character
`noise-retail`	noise	passive	many-small	—	Uninformed retail flow: small random orders resting near the touch
`market-taker`	noise	sweep (MARKET)	cautious	—	Impatient noise trader that takes liquidity
`iceberg-seller`	noise	iceberg	few-large	—	Large orders showing a fifth of their size
`patient-liquidity`	noise	passive (3-15 ticks deep)	own	—	Small orders waiting deep in the book for up to half an hour: book depth
`scalper`	trend	sweep (IOC)	aggressive	trailing stop	Fast momentum trader
`trend-follower`	trend	marketable	cautious	stop	Follows momentum with marketable limits

Preset	Strategy	Execution	Tempo	Protection	Character
`block-taker`	trend	sweep (FOK)	few-large	stop-limit	Takes whole blocks or nothing
`contrarian`	reversion	passive (GTC)	cautious	bracket (OCO)	Fades moves patiently
`auction-player`	reversion	passive	cautious	—	Trades mostly in the opening and closing auctions (ATO/ATC)
`value-investor`	value	passive	cautious	—	Buys below its view of the true value, sells above
`institutional`	value	twap	few-large	—	Large parent orders toward the value, worked in slices over minutes
`news-trader`	news	marketable	aggressive	—	Trades headlines within seconds, in their direction

Between them the presets send every order type and time in force the exchange has: LIMIT, MARKET, IOC, FOK, ICEBERG, STOP, STOP_LIMIT, TRAILING_STOP, OCO; DAY, GTC, ATO, ATC.

12.2.1 Value and news need the market model

The `value` and `news` strategies trade on what `pm-market-sim` publishes: a hidden *true value* per symbol, and news. Without it running, `value-investor`, `institutional` and `news-trader` stand idle — they log on but send nothing. The true values never reach students: only the AI traders, `pm-mm-bot --anchor-sim` and ADMIN keys see them. The headlines are public.

12.3 Running one AI trader

```
pm-ai-trader --id AI001 --preset noise-retail --symbols AAPL,MSFT -v
```

Flag	Default	Meaning
`--id`	—	Participant ID to log on as
`--preset` / `--preset-file`	`noise-retail`	A built-in preset, or your own preset YAML
`--list-presets`	—	Print the built-in presets and exit
`--symbols`	every symbol	Comma-separated symbols to trade
`--seed`	`1`	Same seed, same decisions given the same market
`--alf-host` / `--alf-port`	engine host / deployed `alf_gateway.port`	Where `pm-alf-gwy` listens
`--duration`	`0`	Seconds to run; `0` = until Ctrl+C
`-v`	off	Log a status line a minute and every session change

Stop it with Ctrl+C: it closes its session, and its orders are cancelled.

12.4 Running a swarm

```
pm-ai-swarm --swarm docs/examples/ref_data/s150-nominal-setup/swarm.yaml -v
```

`swarm.yaml` says how many traders, which presets in what mix, which symbols, and how many order actions a second the whole swarm may send:

```
version: 1
agents: {prefix: AI, start: 1, count: 20}
budget: 40            # order actions per second, whole swarm
seed: 1000            # agent i gets seed + i
symbols:
  per_agent: 150      # every agent trades every symbol
composition:          # preset: weight
  noise-retail: 25
  value-investor: 22
  news-trader: 5
  institutional: 8
  scalper: 8
  trend-follower: 10
  contrarian: 10
  iceberg-seller: 4
  market-taker: 4
  block-taker: 2
  auction-player: 2
```

Command-line flags override the file (`--count`, `--budget`, `--workers`, `--presets noise-retail:3,scalper:1`, `--symbols-per-agent`, `--duration`). The operator's swarm runbook has every key, how to size a swarm, the

logs, the daily summaries and what happens when a process goes down.

12.5 Your own preset

Copy a built-in preset and change it. A preset file names its four parts:

```
name: nervous-scalper
description: A scalper with a tight trailing stop and small positions.
strategy: {name: trend, strength: 0.9, sample: 4, urgency: 0.9}
execution: {style: sweep, sweep_type: IOC, cross_ticks: 1}
tempo: {decisions_per_min: 30, size_min: 10, size_max: 50}
risk: {max_position: 200, max_live_orders_per_symbol: 2, max_order_age_sec: 15,
       protection: trailing, protection_pct: 0.005}
```

```
pm-ai-trader --id AI001 --preset-file nervous-scalper.yaml -v
```

In a `swarm.yaml`, a composition key that ends in `.yaml` is a preset file, relative to the `swarm.yaml`. Unknown keys and out-of-range values are refused before anything starts, naming the file and the key.

12.6 Reading its output

With `-v`, each trader (or each swarm worker, for its traders together) logs one line a minute:

```
[AI001] t=120s session=CONTINUOUS actions=96 (0.8/s) live_orders=2 fills=31 rejects=0
```

`actions` counts new orders and cancels. A trader whose orders keep being refused pauses itself for a few seconds (the *reject breaker*); a cancel that lost the race with a fill, or an order in flight when the market closed, does not count. After each close the swarm writes a summary per trader — orders, fills, position, P&L — see the runbook.

12.7 See also

- How the AI Traders Decide — strategies, execution styles, tempo and risk in detail
- Running the AI-Trader Swarm — the operator's runbook
- The Market Model (`pm-market-sim`) — true values and news
- The Market-Maker Bot (`pm-mm-bot`) — liquidity for the AI traders to trade against
- Order Types
- AI Traders & News — classroom exercises

CHAPTER 13

How the AI Traders Decide

Learning objectives

After reading this page you will be able to:

- Follow one AI trader from a decision to the orders it sends
- Explain each strategy, execution style, tempo and risk setting
- Predict how a preset will behave, and write your own

Prerequisites: AI Traders.

13.1 The loop

An AI trader does nothing on a fixed timer. Decisions arrive at random — a Poisson process at the tempo's `decisions_per_min` — so a swarm's orders do not come in bursts. In between, it keeps its house in order. Each time it wakes, in this order:

1. **Housekeeping.** Cancel resting orders older than `max_order_age_sec`, or more than `stale_price_ticks` behind the touch on their own side (a buy order behind the best bid, a sell order behind the best ask). Not in pre-open and the auction calls: there an order waits for the uncross, and whatever is left after it goes at the first sweep of continuous trading.
2. **Slices due** of a `twap` parent order.
3. **Protection.** Place, resize or remove the protective orders for each open position (continuous trading only).
4. **A decision**, if one is due: the strategy names a symbol, a side, an urgency and a size multiplier — an *intent* — or nothing. The execution style turns the intent into orders, within the risk limits.

During continuous trading the decision rate follows the day's U shape: about twice the average after the open and before the close, half of it at midday. A swarm's `budget` scales every trader's rate so the swarm as a whole sends the actions per second it was given.

The trader reads the market from the engine's public feed — book snapshots, trades, the session, halts — and from `pm-market-sim` the true values and the news. It sends nothing for a halted symbol, nothing while the market is closed, and only what each phase allows: resting LIMIT and ICEBERG orders in the call phases, ATO in the opening auction, ATC in the closing auction, everything during continuous trading.

13.2 Strategies: what to trade

A strategy says which way it leans with a *score* from −1 to 1 and buys with probability `(1 + score × strength) / 2`, so it leans without becoming predictable.

Strategy	Looks at	Buys when	Parameters
`noise`	one random symbol	at random (score 0)	`urgency`
`trend`	`sample` random symbols, acts on the strongest signal	its fast average price (30 s) is above its slow one (300 s)	`strength`, `sample`, `urgency`
`reversion`	the same way	the price is below its slow average	`strength`, `sample`, `urgency`
`value`	`sample` random symbols, acts on the widest gap	the price is below its view of the true value by more than `threshold`	`threshold`, `bias_std`, `redraw`, `sample`, `urgency`, `rumour_shift`
`news`	the news touching its symbols	the headline is good news	`lag_sec`, `lag_sigma`, `half_life_sec`, `threshold`, `urgency`, `move`

Trend and reversion need about 30 seconds and 20 prices of a symbol before they have an opinion on it. Both scale the difference by the symbol's own recent volatility, so a 1% move means more in a quiet stock than in a wild one. The stronger the signal, the more urgent the intent.

Value sees each symbol's true value through a bias of its own — a persistent error, drawn per symbol with standard deviation `bias_std` and re-drawn now and then (`redraw`). That is why two value traders disagree, and trade with each other. It acts once the log gap between its view and the price exceeds `threshold`; the wider the gap, the more urgent and the larger the order, up to three times its usual size. When it crosses the spread it pays up to its view of the value less `threshold` — not one tick past the touch — so a jump in the value is priced in by a few orders rather than a thousand one-tick steps. An open rumour shifts its view by `rumour_shift × sentiment × credibility`.

News hears each headline after a delay of its own (lognormal, median `lag_sec`), so news traders do not all hit the book in the same millisecond. From then on its conviction is the headline's |sentiment| — times the credibility for a rumour — halving every `half_life_sec`. While conviction is above `threshold` it trades the headline's symbols in the headline's direction, urgently and in up to three times its usual size, paying up to `move × conviction` (a log move) beyond the price it saw when it first acted on the headline. A confirmation turns a rumour into news known to be true; a retraction leaves an impulse the other way, as large as the rumour was believable.

13.3 Execution styles: how to trade it

Style	Sends	Parameters
`passive`	A LIMIT `offset_ticks` behind the touch (0 joins it), or a random depth up to `offset_max`, most of them near the touch	`offset_ticks`, `offset_max`, `tif`
`marketable`	A LIMIT `cross_ticks` through the touch; what does not fill rests	`cross_ticks`, `tif`
`sweep`	A MARKET, IOC or FOK order against the other side	`sweep_type`, `cross_ticks`
`iceberg`	An ICEBERG behind the touch showing `visible_fraction` of its size	`offset_ticks`, `visible_fraction`
`twap`	A parent order cut into `slices` child orders over `horizon_sec`	`slices`, `horizon_sec`, `child_style`

Any style becomes marketable when the intent's urgency reaches `urgency_cross`. A crossing order (marketable, IOC, FOK) is priced at the strategy's limit when it gives one (`value` and `news` do), otherwise `cross_ticks` past the touch. In the call phases — pre-open and the auctions — a trader takes part in a decision with probability `auction_participation`, and then sends a limit order straddling the reference price (with TIF ATO or ATC in the auctions), so that buyers and sellers overlap at the uncross. Prices are kept inside the symbol's price collar and quantities inside its order limits: an AI trader does not send what the engine would refuse.

Three more execution settings, off unless a preset sets them:

Setting	Effect
`smp`	The self-match prevention mode sent on every order (`NONE`, `CANCEL_AGGRESSOR`, `CANCEL_RESTING`, `CANCEL_BOTH`); unset, the gateway's default applies
`amend_fraction`	That share of the orders the risk rules would cancel is amended instead: one left behind is re-priced to its own side's touch (losing priority), one still at the touch is trimmed to half its unfilled quantity (keeping priority)
`combo_fraction`	That share of continuous-trading decisions is sent as a small all-or-none COMBO: the intended order and the opposite side of another of the trader's symbols, both at the touch, a quarter of the size

13.4 Tempo: how often and how big

Tempo	Decisions per minute	Order size	Sizes
`many-small`	30	1–25	mostly small
`aggressive`	20	20–120	even
`cautious`	6	10–60	even
`few-large`	2	150–700	mostly large

A preset can give its own instead: `tempo: {decisions_per_min: 12, size_min: 5, size_max: 80, size_distribut small-heavy}` (`balanced`, `small-heavy` or `block-heavy`). The strategy's size multiplier applies on top.

13.5 Risk: what it will not do

Setting	Effect
`max_position`	No order that could take the position past ± this many shares, counting open orders as if filled
`max_live_orders_per_symbol`	At most this many working orders per symbol
`max_order_age_sec`	Cancel resting orders older than this
`stale_price_ticks`	Cancel resting orders this far behind their own side's touch
`protection`, `protection_pct`	Protective orders for open positions, `protection_pct` of the price away

`protection` is one of `none`, `stop` (a STOP order), `stop_limit`, `trailing` (a TRAILING_STOP) or `bracket` (an OCO of a take-profit LIMIT and a STOP). It follows the position: a new fill resizes it, a flat position removes it.

A trader that gets 25 orders refused within 10 seconds pauses for 5 seconds — the *reject breaker* — so a misconfigured preset cannot flood the exchange with refused orders. A cancel that lost the race with a fill, or an order in flight when the market closed, does not count.

13.6 Writing a preset

Start from the closest built-in preset in `src/edumatcher/ai_trader/presets/` and change one part at a time:

You want	Change
More flow	`tempo`
Takes liquidity instead of providing it	`execution.style: sweep` or `marketable`
Holds bigger positions	`risk.max_position`
Reacts to news faster	`strategy.lag_sec` (news), `execution.urgency_cross`
Disagrees more with other value traders	`strategy.bias_std` (value)

Every parameter has a range; a value outside it is refused with the file and the key named. AI Traders shows how to run your preset.

13.7 The other bot: `pm-mm-bot`

`pm-ai-trader` trades like a directional participant. The market-maker bot has a different job: it posts two-sided quotes on many symbols from one participant ID. Run one with the swarm — AI traders alone make a thin, jumpy book.

CHAPTER 14

Positions and P&L

Learning objectives

After reading this page you will understand:

- What P&L and clearing mean in an exchange context
- How long and short positions are tracked and how VWAP average cost works
- The difference between realized and unrealized P&L
- How the `pm-clearing` process subscribes to trade and lifecycle events
- The SQLite database schema and what each table stores
- How to query clearing data using every `pm-clearing-cli` command verb
- How to inspect end-of-day marks and gateway session history
- A practical cookbook for answering clearing-house questions from the command line

Prerequisite: Complete Your First Trade.

14.1 What this page covers

P&L stands for **Profit and Loss** — the running tally of how much money each trader has made or lost. **Clearing** is the process of settling trades after they execute: confirming who owes what to whom, updating account balances, and recording the transaction history.

In a real exchange, clearing involves counterparty risk management, margin calls, and settlement cycles (T+1 or T+2). During the day EduMatcher simplifies this to real-time P&L accounting with no credit risk — every trade settles instantly.

EduMatcher v2 uses a **two-component clearing system**:

Component	Role
`pm-clearing`	Long-running subscriber — reads every `trade.executed` event, maintains in-memory positions and P&L, and writes batched results to `clearing.db` (SQLite)

Component	Role
`pm-clearing-cli`	One-shot query tool — reads from `clearing.db` and prints human-friendly or machine-readable output without SQL

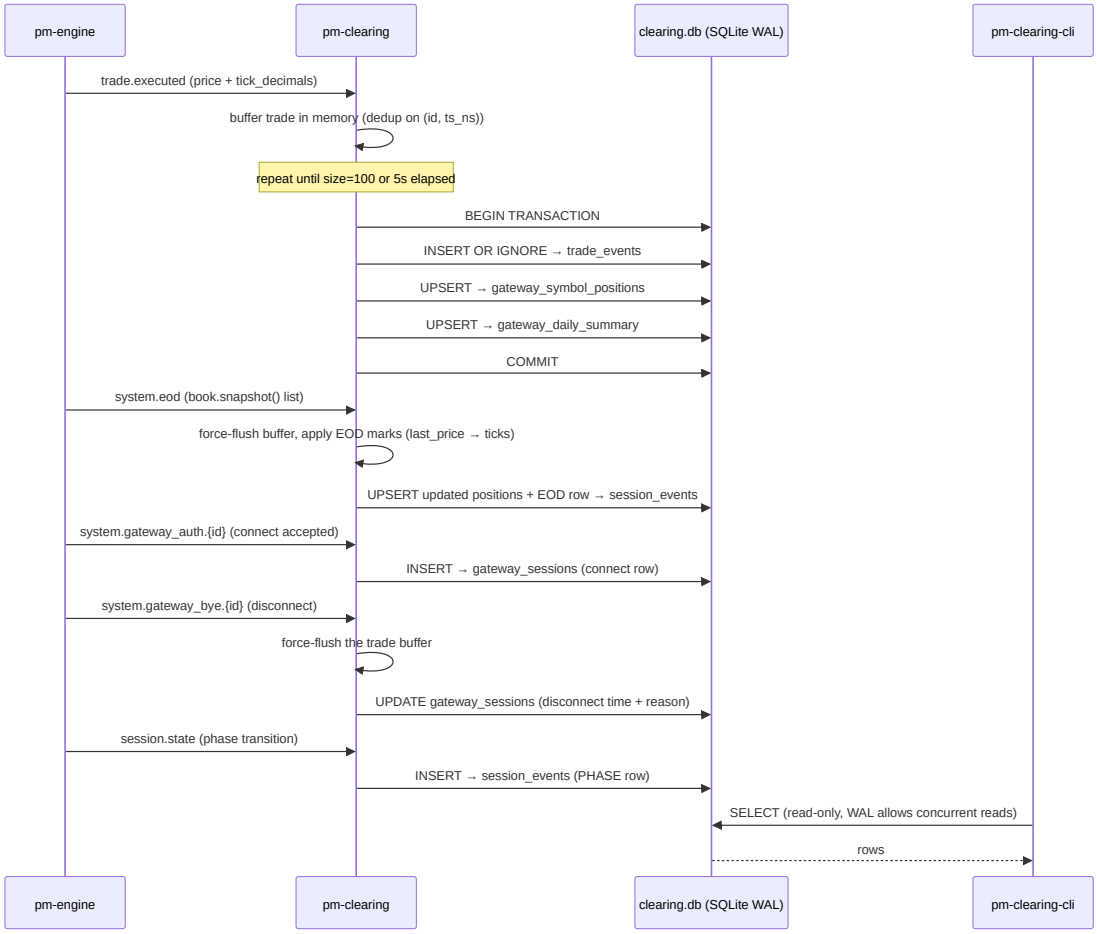
14.2 Starting the clearing process

```
pm-clearing
```

No arguments are required. The process connects to the engine's PUB socket (port 5556), creates or opens `data/clearing.db`, and begins tracking P&L. It also binds a health socket (`127.0.0.1:5563` by default) that answers `pm-clearing-cli health` pings. It can be started before or after trading begins — it will pick up all trades from the moment it subscribes.

Warm start on restart. If `clearing.db` already contains positions, `pm-clearing` rebuilds its in-memory ledger from `gateway_symbol_positions` before it subscribes, so a restart *accumulates* onto the durable position state instead of resetting every position to flat and overwriting it. This means `pm-clearing` can be stopped and restarted mid-session without corrupting positions or P&L.

14.2.1 Message flow



14.2.2 Optional arguments

```
pm-clearing [OPTIONS]

  --datapath PATH        Data directory or explicit .db file path
  --db-name NAME         SQLite filename within data dir (default: clearing.db)
  --flush-size N         Max buffered trades before flush (1..100, default: 100)
  --flush-interval SEC   Max seconds between flushes (>=0.1, default: 5)
  --print-every N        Print P&L summary every N trades (0 = never, default: 100)
  --retention-days N     Prune trade_events rows older than N days on startup,
                         at every close and on each system.eod (default: 90) —
                         see the note under Retention below for what `0` does
                         (not quite "disable pruning", despite the real
                         `--help` text's wording)
  --log-level LEVEL      Explicit log level: CRITICAL, ERROR, WARNING,
                         INFO, DEBUG (default: WARNING)
  --log-target TARGET    Where operational logs go: server (default,
                         auto-detected pm-log-srv), stdout, or file
  --log-file PATH        Log file path (required with --log-target file)
  --log-failover-timeout SEC
                         Grace window before falling back to a local log file
                         when pm-log-srv is unreachable (default: 30)
  -v, --verbose          Increase verbosity (-v: INFO, -vv: DEBUG)
  -q, --quiet            Reduce log output to warnings/errors
  --sql-trace            Log executed SQLite SQL statements (off by default)
  --version              Show version and exit
  --help                 Show help and exit
```

Trade-date bucketing and the session timezone

Each trade is assigned a `trade_date` (used by `gateway_daily_summary`, the `daily`/`dates` verbs, and the EOD row) based on the calendar day of its timestamp in the deployed config's `locale.session_timezone`. The default is `Europe/Stockholm`. Set it to the exchange's local timezone so an evening session — or any session that straddles 00:00 UTC — stays in a single `trade_date` bucket instead of being split across two days.

14.2.3 What pm-clearing subscribes to

`pm-clearing` subscribes to these topics on the engine PUB socket:

Topic	Required	Action
`trade.executed`	Yes	De-duplicate on `(id, ts_ns)`, buffer, apply to ledger, batch-write to DB
`system.eod`	No (secondary)	Force-flush, apply EOD marks from the book snapshot, write `session_events` row
`system.gateway_auth.{id}`	No (secondary)	Record a `gateway_sessions` connect row when a gateway's auth is accepted

Topic	Required	Action
`system.gateway_bye.{id}`	No (secondary)	Force-flush the whole trade buffer, update the `gateway_sessions` row with the disconnect
`session.state`	No (secondary)	Write a `PHASE` row to `session_events` recording the new and previous session state

The secondary subscriptions add contextual information without affecting the core P&L accuracy. If they are not received (for example, engine was killed hard rather than gracefully shut down), position data is still complete.

Gateway lifecycle is read from the PUB broadcasts

`system.gateway_connect` / `system.gateway_disconnect` are **gateway → engine** messages on the engine's PULL socket — they are never seen by a PUB subscriber. The engine broadcasts the corresponding lifecycle events on PUB as `system.gateway_auth.{id}` (on connect) and `system.gateway_bye.{id}` (on disconnect), and `pm-clearing` records sessions from those. The inbound topic names are still accepted for direct-injection tooling and tests.

`system.eod` — end-of-day finalisation

On receipt, `pm-clearing`:

1. **Force-flushes** any buffered trades immediately, bypassing the 100-trade and 5-second thresholds.
2. Applies EOD **mark-to-market**. The `system.eod` message carries a list of `book.snapshot()` dicts, so `pm-clearing` reads each book's `last_price` (a display price) — or the top-of-book `bids[0]`/`asks[0]` mid when no trade occurred — and converts it to integer ticks with `to_ticks(...)` to update `mark_price` and `unrealized_pnl` in every open position.
3. Writes the re-marked positions to `gateway_symbol_positions` only. `gateway_daily_summary.end_unrealized_` is **not** touched by the EOD pass — it keeps the value from the last trade flush.
4. Inserts an `EOD` sentinel row into `session_events` with the timestamp and the mark prices applied (in ticks). This lets `pm-clearing-cli eod` report exact session-close times.

`system.gateway_auth.{id}` / `system.gateway_bye.{id}` — gateway lifecycle

On an **accepted connect** (`system.gateway_auth.{id}` with `accepted=true`), `pm-clearing` inserts a row into `gateway_sessions` recording `gateway_id` and the ingestion timestamp; a refused auth opens no session. On **disconnect** (`system.gateway_bye.{id}`), it updates that row with `disconnected_at_ns` and the disconnect reason, and immediately force-flushes the entire trade buffer (not just that gateway's trades) so no fill is lost before the engine processes its order cancellations.

`session.state` — session phase transitions

On receipt, `pm-clearing` inserts a `PHASE` row into `session_events` recording the new state and the previous state (e.g. `CONTINUOUS` → `CLOSING_AUCTION`). This is purely informational — it does not affect P&L, positions, or the EOD mark — and gives `session_events` a running record of every phase transition the engine went

through during the session. See session_events below for the full event-type reference, including which types are (and are not) surfaced by pm-clearing-cli.

14.3 Data folder location

pm-clearing writes to clearing.db in the same data directory used by all other EduMatcher processes:

Running mode	Default location	Override
Source checkout	<repo>/src/data/clearing.db	EDUMATCHER_DATA_DIR
Installed (pipx)	~/.local/share/edumatcher/clearing.db	EDUMATCHER_DATA_DIR

Pass --datapath to use a different directory or file:

```
# Use a custom directory
pm-clearing --datapath ~/trading/sessions/morning

# Use an explicit db file
pm-clearing --datapath ~/trading/clearing_2026.db
```

14.4 SQLite database schema

pm-clearing creates and maintains six tables and two views in clearing.db. All tables are created idempotently on startup so the same DB file can be re-opened safely across process restarts.

14.4.1 Table overview

Table	Purpose
trade_events	Append-only fact table; one row per trade.executed event
gateway_symbol_positions	Running position state; one row per (gateway_id, symbol)
gateway_daily_summary	Daily rollup aggregates; one row per (trade_date, gateway_id, symbol)
session_events	Clearing-significant events (EOD, PHASE session-state transitions, plus GAP integrity alarms)
gateway_sessions	Gateway connect / disconnect history
position_close_snapshot	Every position as it stood at each session close; the input of the end-of-day settlement

14.4.2 trade_events

Append-only audit log. Populated by pm-clearing from trade.executed using INSERT OR IGNORE, idempotent on the durable trade ID id.

The engine's trade `id` is durable across restarts, formatted as `run_seq-counter` (for example, `000042-000000001`), and is the sole database identity. A genuine duplicate delivery is therefore de-duplicated even if its delivery timestamp differs. Databases created before durable IDs are rejected and must be recreated; the system does not migrate historical short IDs.

Column	Type	Description
`id`	TEXT (PK)	Durable engine trade id, formatted as `run_seq-counter`
`ts_ns`	INTEGER	Engine event timestamp in nanoseconds
`trade_date`	TEXT	Session-timezone date (`YYYY-MM-DD`) derived from `ts_ns` (default `Europe/Stockholm`; see `locale.session_timezone`)
`symbol`	TEXT	Instrument symbol
`quantity`	INTEGER	Matched trade size
`price`	INTEGER	Execution price in ticks
`tick_decimals`	INTEGER	Symbol precision (`10^-d`)
`buy_order_id`	TEXT	Buy-side order reference
`sell_order_id`	TEXT	Sell-side order reference
`buy_gateway_id`	TEXT	Gateway credited with the buy fill
`sell_gateway_id`	TEXT	Gateway credited with the sell fill
`aggressor_side`	TEXT	`BUY`, `SELL`, `AUCTION` (uncross prints, where both sides rested), or NULL for legacy payloads
`ingest_ts_ns`	INTEGER	Ingestion timestamp (local wall clock)

Retention: rows older than `--retention-days` (default 90, compared against the UTC date) are deleted on startup, at every close of the market (`session.state CLOSED`), on each `system.eod`, and on demand via `pm-clearing-cli prune`.

`-retention-days 0` behaves differently depending on which trigger fires

The three automatic triggers are not consistent about `0`. On **startup** and on **every `system.eod`**, pruning runs unconditionally — `0` does **not** disable it there, it deletes every row dated before today (UTC). On **market close** (`session.state` → `CLOSED`) specifically, pruning is skipped entirely when `--retention-days` is `0` — the one trigger where `0` genuinely means "keep everything". `pm-clearing-cli prune` is unaffected either way: its own `--days` rejects `0` outright (`--days` must be `>= 1`). If you want to reliably keep every row forever regardless of which trigger fires, use a large value (e.g. `36500`) rather than relying on `0`.

14.4.3 `gateway_symbol_positions`

Live running state for every `(gateway_id, symbol)` key seen so far. Upserted on every flush (`INSERT … ON CONFLICT(gateway_id, symbol) DO UPDATE`). On startup, `pm-clearing` warm-starts its in-memory ledger from this table (see Warm start on restart), so a restart resumes from the persisted positions rather than overwriting them from flat.

Column	Type	Description
`gateway_id`	TEXT	Gateway identifier
`symbol`	TEXT	Instrument symbol
`net_qty`	INTEGER	Signed net quantity (+ long, − short)
`avg_cost`	REAL	VWAP average entry cost, in ticks (fractional because of VWAP division; `pm-clearing-cli` divides by `10^tick_decimals` for display)
`realized_pnl`	REAL	Cumulative realized P&L, in ticks
`unrealized_pnl`	REAL	Current open mark-to-market, in ticks
`mark_price`	INTEGER	Latest trade price in ticks; updated on `system.eod`
`tick_decimals`	INTEGER	Precision for this symbol
`buy_qty`	INTEGER	Cumulative buy-side filled quantity
`sell_qty`	INTEGER	Cumulative sell-side filled quantity
`buy_notional`	INTEGER	Cumulative buy-side notional in ticks
`sell_notional`	INTEGER	Cumulative sell-side notional in ticks
`last_trade_ts_ns`	INTEGER	Latest trade timestamp for this key
`updated_ts_ns`	INTEGER	Flush timestamp

14.4.4 `gateway_daily_summary`

Daily incremental aggregates. Updated in the same transaction as `gateway_symbol_positions`.

Column	Type	Description
`trade_date`	TEXT	Session-timezone date (`YYYY-MM-DD`); default `Europe/Stockholm` (see `locale.session_timezone`)
`gateway_id`	TEXT	Gateway identifier
`symbol`	TEXT	Instrument symbol
`traded_qty`	INTEGER	Daily total filled quantity for this key
`traded_notional`	INTEGER	Daily total notional in ticks
`buy_qty`	INTEGER	Daily buy-side filled quantity
`sell_qty`	INTEGER	Daily sell-side filled quantity
`buy_notional`	INTEGER	Daily buy-side notional in ticks
`sell_notional`	INTEGER	Daily sell-side notional in ticks
`net_amount`	INTEGER	`buy_notional - sell_notional`
`realized_pnl`	REAL	Daily realized P&L contribution, in ticks
`end_net_qty`	INTEGER	Net quantity at last flush for this date
`end_avg_cost`	REAL	Average cost at last flush, in ticks
`end_unrealized_pnl`	REAL	Unrealized P&L at the last trade flush, in ticks (not re-marked at EOD)
`tick_decimals`	INTEGER	Symbol precision
`last_trade_ts_ns`	INTEGER	Latest trade timestamp
`updated_ts_ns`	INTEGER	Flush timestamp

14.4.5 `session_events`

Append-only log of clearing-significant events. Three event types are written:

`event_type`	Written when	`payload_json`
`EOD`	`system.eod` received on graceful engine shutdown	`{"eod_marks": {symbol: price_ticks, ...}, "symbols_count": N}`
`PHASE`	`session.state` received (session phase transition)	`{"state": ..., "prev_state": ...}`

event_type	Written when	payload_json
GAP	The engine's trade-id suffix jumps forward within a run, i.e. the lossy PUB feed dropped one or more trades	{"run_seq": R, "last_seq": L, "next_seq": N, "missing_trades": M}

Column	Type	Description
id	INTEGER (autoincrement)	Surrogate key
event_type	TEXT	EOD, PHASE, or GAP
ts_ns	INTEGER	Ingestion timestamp
trade_date	TEXT	Session-timezone date derived from ts_ns (default Europe/Stockholm; see locale.session_timezone)
payload_json	TEXT	Event-specific JSON (see table above)

EOD rows are surfaced by pm-clearing-cli eod. PHASE and GAP are not currently surfaced by any pm-clearing-cli verb — GAP is a durable integrity alarm, recorded so a dropped-trade event is visible after the fact rather than silently corrupting positions; query both directly from session_events by event_type. A trade id that is re-sent is silently ignored by the archive (INSERT OR IGNORE on the trade-id primary key); no alarm row is written.

14.4.6 position_close_snapshot

One row per position at each session close, **whether or not it traded that day**. It exists because gateway_daily_summary only has rows for symbols a gateway *traded*, so a position that was merely held while the price moved would otherwise be invisible to the end-of-day settlement (Closing the Day). It is written when the session reaches CLOSED (after the buffered trades are flushed) and on system.eod when the engine shuts down gracefully, replacing that date's rows — **unless the date's snapshot is final**: the date's latest session phase is CLOSED, the market was open earlier that day, and a snapshot exists. A final snapshot is the real close and may already be settled, so neither a shutdown nor the CLOSED an engine restart announces after the close replaces it. Rows are small and **are not pruned** with trade_events.

Column	Type	Description
trade_date	TEXT	Trading date of the close (session timezone)
gateway_id, symbol	TEXT	The position (primary key with trade_date)
net_qty	INTEGER	Signed quantity at the close (negative = short; 0 = flat)

Column	Type	Description
`realized_pnl`	REAL	**Cumulative** realized P&L, in ticks
`unrealized_pnl`	REAL	Unrealized P&L at `mark_price`, in ticks
`mark_price`	INTEGER	The closing mark, in ticks (the last trade price)
`tick_decimals`	INTEGER	Decimal places of the symbol (ticks → currency)

Query with `pm-clearing-cli snapshots`.

14.4.7 `gateway_sessions`

One row per accepted gateway connection. Updated with disconnect time when the engine's `system.gateway_bye.`
broadcast is received.

Column	Type	Description
`gateway_id`	TEXT	Gateway identifier
`connected_at_ns`	INTEGER	Ingestion timestamp of connect event
`disconnected_at_ns`	INTEGER	Ingestion timestamp of disconnect; NULL if session still open
`disconnect_reason`	TEXT	Reason string from disconnect message; NULL if not provided

Query with `pm-clearing-cli sessions`.

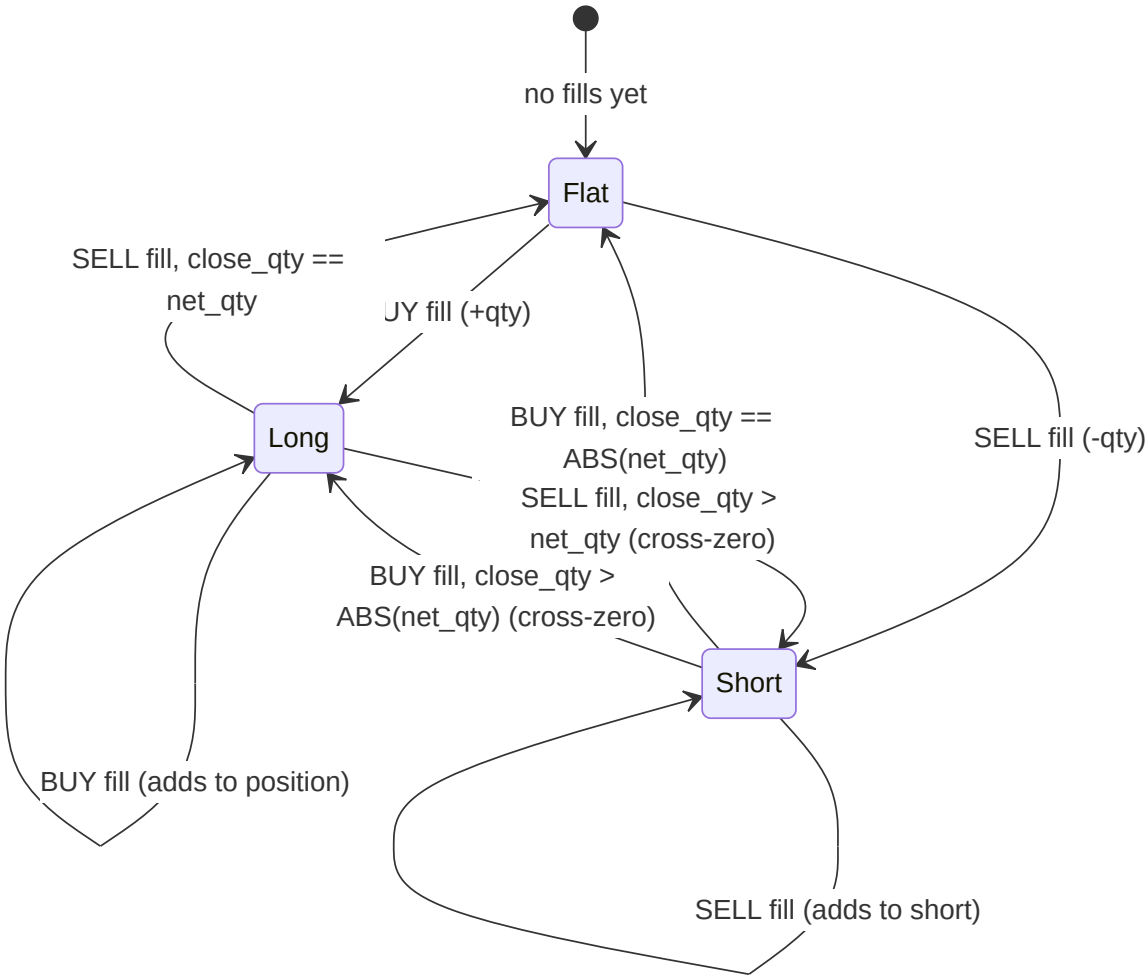
14.5 Position tracking

For each `(gateway_id, symbol)` pair, `pm-clearing` tracks:

Field	Description
`net_qty`	Net quantity held. Positive = long, negative = short, zero = flat.
`avg_cost`	Volume-weighted average entry price (VWAP).
`realized_pnl`	Accumulated P&L from trades that reduced the position.
`unrealized_pnl`	Paper P&L on the remaining open position.

Field	Description
`mark_price`	Latest observed trade price, used to value the open position.

14.5.1 Position state machine



Realized P&L is calculated on every state transition that reduces an open position.

14.6 VWAP average cost

When you build a position through multiple trades at different prices, the **volume-weighted average price (VWAP)** gives your blended entry cost:

$$\text{avg_cost}_{\text{new}} = \frac{\text{avg_cost}_{\text{old}} \times |\text{net_qty}_{\text{old}}| + \text{fill_price} \times \text{fill_qty}}{|\text{net_qty}_{\text{new}}|}$$

This is recalculated on every fill that increases an existing position.

14.7 Realized P&L

Realized P&L is profit or loss that is locked in — it comes from trades that **reduce** your position.

Closing a long position (via SELL):

$$\text{realized+} = (\text{fill_price} - \text{avg_cost}) \times \text{closed_qty}$$

Closing a short position (via BUY):

$$\text{realized+} = (\text{avg_cost} - \text{fill_price}) \times \text{closed_qty}$$

When a fill crosses zero (e.g. closes a 100-share long and opens a 30-share short in one trade), P&L is realized on the full close and `avg_cost` is reset to the fill price for the new-side position.

14.8 Unrealized P&L

Unrealized P&L is the theoretical value of the remaining open position at the current mark price:

$$\text{unrealized} = \text{net_qty} \times (\text{mark_price} - \text{avg_cost})$$

For a short position `net_qty` is negative, so a falling mark price produces positive unrealized P&L.

14.9 Worked example

Step	Event	Calculation	net_qty	avg_cost	realized_pnl
1	BUY 10 @ 100	Open long	+10	100.00	0
2	BUY 10 @ 110	Add to long. avg = (1000+1100)/20	+20	105.00	0
3	SELL 20 @ 115	Full close. realized = (115−105)×20	0	0	200
4	SELL 15 @ 108	Open short from flat	−15	108.00	200
5	BUY 20 @ 105	Close 15 short, open 5 long. realized += (108−105)×15	+5	105.00	245

14.10 Position arithmetic in depth

This section walks through every position transition step by step, starting from scratch. It is written for readers who find the formulas above abstract or who are puzzled by how cross-zero trades work internally.

14.10.1 What a position actually is

A position is EduMatcher's record of three things for every `(gateway_id, symbol)` pair:

1. **How many units are currently held** (`net_qty`) — positive means long (you own them), negative means short (you owe them), zero means flat (no exposure).
2. **At what blended price they were acquired** (`avg_cost`) — the VWAP of all fills that built the current open position.
3. **How much profit has already been locked in** (`realized_pnl`) — gains and losses from fills that *reduced* the position.

Every `trade.executed` message produces exactly two fill applications: one crediting the buyer's gateway and one crediting the seller's. Each application runs through the same position-update logic described below.

14.10.2 The four fill outcomes

There are four things that can happen when a fill is applied to a position:

Outcome	When it happens	Effect on avg_cost	Realized P&L?
Open	position is flat	set to fill price	No
Add	fill is same direction as existing position	recalculated (VWAP)	No
Reduce / Close	fill opposes existing position, does not exceed it	unchanged	Yes, on closed qty
Cross-zero	fill opposes existing position and exceeds it	reset to fill price (for new side)	Yes, on closed qty only

14.10.3 Opening a position (flat → long or flat → short)

When `net_qty = 0`, the position is flat and any fill simply opens it.

Example — BUY 10 @ 100:

```
Before:  net_qty = 0,  avg_cost = 0.00,  realized_pnl = 0

Step 1:  net_qty  = 0 + 10 = 10
Step 2:  avg_cost = 100.00      (fill price becomes the entry cost)

After:   net_qty = +10,  avg_cost = 100.00,  realized_pnl = 0
```

Nothing is realized here. You now own 10 units at an average cost of 100.

14.10.4 Adding to an existing position (VWAP update)

When a fill is in the **same direction** as the current position — buying while already long, or selling while already short — the position grows and `avg_cost` is recalculated as a volume-weighted average.

Example — BUY 10 @ 110 (already long 10 @ 100):

```
Before:  net_qty = +10,  avg_cost = 100.00

existing_notional = 100.00 × 10 = 1,000
incoming_notional = 110.00 × 10 = 1,100

net_qty  = 10 + 10 = 20
avg_cost = (1,000 + 1,100) / 20 = 105.00

After:   net_qty = +20,  avg_cost = 105.00,  realized_pnl = 0
```

The VWAP of 105 reflects the blended cost of all 20 shares: 10 bought at 100 and 10 bought at 110.

14.10.5 Reducing a position (partial or full close)

When a fill **opposes** the current position but does not exceed it in size, the position shrinks. P&L is realized on the closed portion only. `avg_cost` is **not** changed — it reflects your entry basis and stays fixed until you either go flat or cross zero.

Example — SELL 5 @ 115 (long 20 @ 105):

```
Before:  net_qty = +20,  avg_cost = 105.00,  realized_pnl = 0

close_qty = min(5, 20) = 5

realized_delta = (fill_price − avg_cost) × close_qty
               = (115 − 105) × 5 = 50

net_qty = 20 − 5 = 15          (still long)
avg_cost unchanged at 105.00   (entry basis of remaining shares)

After:   net_qty = +15,  avg_cost = 105.00,  realized_pnl = 50
```

The $50 is now permanent regardless of what price does next.

Full close — SELL 15 @ 115 (long 15 @ 105):

```
close_qty = min(15, 15) = 15

realized_delta = (115 − 105) × 15 = 150

net_qty  = 15 − 15 = 0
avg_cost = 0.00    (reset to zero when flat)

After:   net_qty = 0,  avg_cost = 0.00,  realized_pnl = 200
```

Position is flat. `avg_cost` resets to zero because there is nothing left to track an entry basis against.

14.10.6 Cross-zero: the confusing case

Cross-zero happens when a single fill is large enough to close the entire existing position *and* still have excess quantity left over, which opens a new position on the **opposite** side — all in one step.

Think of it as two things happening at once:

1. The fill closes the existing position (realizing P&L on the full open quantity).
2. The leftover quantity opens a brand-new position on the other side at the fill price.

Long-side example: SELL 25 @ 115 when long 15 @ 105

```
Before:  net_qty = +15,  avg_cost = 105.00,  realized_pnl = 0
```

Step 1 — how much closes the long?

```
open_qty  = abs(+15) = 15
close_qty = min(25, 15) = 15    ← only 15 units can close a 15-unit long
excess    = 25 − 15 = 10        ← these 10 open a new short
```

Step 2 — realize P&L on the closed portion:

```
realized_delta = (fill_price − avg_cost) × close_qty
               = (115 − 105) × 15 = 150
```

Step 3 — update net_qty:

```
net_qty = +15 + (−25) = −10    ← positive → zero → negative: crossed zero
```

Step 4 — the position flipped sides, so reset avg_cost to the fill price:

```
avg_cost = 115.00    ← entry cost of the new 10-unit short
```

Final state:

```
net_qty       = −10
avg_cost      = 115.00
realized_pnl  = 150
unrealized_pnl = −10 × (115 − 115) = 0   (mark = fill price, no gain yet)
```

The 10-unit short contributes *zero* realized P&L at this moment. Its P&L will only become realized when a future buy reduces or closes it.

Realized P&L is only for the closing portion

In a cross-zero fill, `close_qty = min(fill_qty, open_qty)` caps the realized calculation at the original position size. The excess quantity that opens the new side starts with zero realized P&L. This prevents double-counting: the new position's entry cost is the fill price, so any future P&L from it starts from that baseline.

Short-side example: BUY 30 @ 105 when short 20 @ 108

The mirror image applies when crossing zero from a short position.

```
Before:  net_qty = −20,  avg_cost = 108.00,  realized_pnl = 0
```

Step 1 — how much closes the short?

```
open_qty  = abs(−20) = 20
close_qty = min(30, 20) = 20
excess    = 30 − 20 = 10
```

Step 2 — realize P&L on the short close:

```
realized_delta = (avg_cost − fill_price) × close_qty
               = (108 − 105) × 20 = 60
```

For a short, falling price is profit, so the formula subtracts in the opposite order: `avg_cost − fill_price`.

Step 3 — update net_qty:

```
net_qty = −20 + 30 = +10    ← negative → zero → positive: crossed zero
```

Step 4 — reset avg_cost for new long:

```
avg_cost = 105.00
```

Final state:

```
net_qty       = +10
avg_cost      = 105.00
realized_pnl  = 60
```

14.10.7 How the code tells the three outcomes apart

When a fill opposes an existing position, the code applies `net_qty += signed_qty` first, then reads the resulting sign to decide what happened:

Scenario (long 15, SELL N)	`net_qty` after	Sign check result	Outcome
SELL 5 (partial close)	+10	still positive, fill was SELL → no match	avg_cost unchanged, partial close
SELL 15 (full close)	0	zero → first `if net_qty == 0` branch fires	avg_cost reset to 0
SELL 25 (cross-zero)	−10	negative and fill was SELL → **match**	avg_cost reset to fill price

The sign check reads: *"after the update, does `net_qty` agree with the fill direction?"* The only way that can be true when entering the reduce/close branch is if the fill exceeded the open quantity — i.e., it crossed zero.

Checking **after** the update is intentional and precise. Before the update, the position is on the opposing side by definition (that is the condition that got us into this branch). After the update, a sign that agrees with the fill direction is the unambiguous fingerprint of a cross-zero event.

14.10.8 Unrealized P&L after every fill

After every fill — regardless of outcome — unrealized P&L is recalculated:

$$\text{unrealized} = \text{net_qty} \times (\text{mark_price} - \text{avg_cost})$$

`mark_price` is set to the latest fill price after each trade. For a flat position (`net_qty = 0`) unrealized P&L is always zero.

For a **short** position `net_qty` is negative, so the formula naturally produces positive unrealized P&L when the price falls below `avg_cost`:

```
net_qty = −10,  avg_cost = 115,  mark_price = 110
unrealized = −10 × (110 − 115) = −10 × (−5) = +50
```

14.10.9 All five outcomes side by side

The table below shows a single gateway trading `AAPL` through all five position transitions in sequence:

#	Fill	open_qty before	close_qty	Outcome	net_qty	avg_cost	realized_pnl
1	BUY 10 @ 100	0	—	Open long	+10	100.00	0
2	BUY 10 @ 110	—	—	Add (VWAP)	+20	105.00	0
3	SELL 5 @ 115	20	5	Partial close	+15	105.00	50
4	SELL 15 @ 115	15	15	Full close	0	0.00	200
5	SELL 15 @ 108	0	—	Open short (from flat)	−15	108.00	200
6	BUY 20 @ 105	15	15	Cross-zero: close 15 short + open 5 long	+5	105.00	245

Row 6 is cross-zero: the BUY 20 closes 15 units of the short (realizing `(108−105)×15 = 45`) and opens a new 5-unit long at 105, accounting for the cumulative jump from 200 to 245.

14.11 Using pm-clearing-cli

`pm-clearing-cli` provides verb-based access to all clearing data without writing SQL. It always reads from the same `clearing.db` used by `pm-clearing`. For a process-level command matrix and operational notes, see Processes — pm-clearing-cli.

```
pm-clearing-cli [GLOBAL_OPTIONS] <verb> [verb-options]

Global options:
  --datapath PATH      Data directory or explicit .db file path
  --db-name NAME       SQLite filename if datapath is a directory
  --format FMT         table | json | csv  (default: table)
  --no-header          Suppress the header row (table and csv)
  --raw-output         Disable normalization and show raw tick-unit values
  --version
  --help
```

`--format`, `--no-header` and `--raw-output` are global options and must come **before** the verb (`pm-clearing-cli --format json gateways`); the verbs themselves do not accept them.

14.11.1 Command verb reference

Verb	Primary source table	What it returns	Key options
`gateways`	`gateway_pnl_totals` view	One row per gateway with total realized, unrealized, and total P&L	`--gateway`, `--limit`
`positions`	`gateway_symbol_positions`	Live position state for every `(gateway, symbol)` key	`--gateway`, `--symbol`, `--limit`
`pnl`	`gateway_symbol_positions`	Realized / unrealized / total P&L, one row per `(gateway, symbol)`	`--gateway`, `--symbol`, `--limit`
`daily`	`gateway_daily_summary`	Daily rollup totals and end-of-day snapshots	`--gateway`, `--symbol`, `--date`, `--from`, `--to`, `--limit`
`trades`	`trade_events`	Raw trade-level audit log	`--gateway`, `--symbol`, `--date`, `--from`, `--to`, `--limit`
`exposure`	`gateway_symbol_positions`	Net and gross notional exposure, sorted by size	`--gateway`, `--symbol`, `--sort`, `--limit`
`symbols`	`gateway_daily_summary` + `gateway_symbol_positions`	Symbol-level cleared volume, notional, P&L, and open position	`--date`, `--from`, `--to`, `--sort`, `--limit`
`dates`	`trade_events` / `daily_exchange_totals`	Available trading dates; add `--with-totals` for per-date volume and net amount	`--gateway`, `--symbol`, `--from`, `--to`, `--with-totals`, `--limit`

Verb	Primary source table	What it returns	Key options
`health`	`pm-clearing` process + health socket; with `--extended`, the whole DB	Liveness of `pm-clearing`; with `--extended`, DB integrity and read/write access; with `--extended -v`, row counts, last trade/flush time and journal mode	`--extended`, `-v`, `-q`
`reconcile`	`trade_events` vs `gateway_daily_summary`	Discrepancies between raw facts and aggregates (both sides), including keys present only in the summaries	`--gateway`, `--symbol`, `--from`, `--to`, `--retention-days`
`sessions`	`gateway_sessions`	Gateway connect and disconnect history written from the engine's `system.gateway_auth` / `system.gateway_bye` broadcasts	`--gateway`, `--from`, `--to`, `--connected-only`, `--limit`
`snapshots`	`position_close_snapshot`	Every position saved at each session close (quantity, cumulative realized and unrealized P&L, closing price)	`--gateway`, `--symbol`, `--date`, `--from`, `--to`, `--limit`
`eod`	`session_events`	End-of-day sentinel rows written on `system.eod`, including mark prices applied	`--from`, `--to`, `--limit`
`prune`	`trade_events`	Deletes rows older than N days (default 90); write-access verb (does not VACUUM)	`--days`, `--dry-run`

14.11.2 gateways — live gateway P&L totals

Returns the cumulative P&L and net position for every gateway that has traded. A clearing house uses this as the **top-level risk snapshot**: it answers "who is exposed and by how much" in a single query, without needing to filter by symbol or date.

Option	Type	Default	Description
`--gateway GW_ID`	string	all gateways	Return only this gateway
`--limit N`	integer	1000	Maximum rows returned

```
# All gateways
pm-clearing-cli gateways

# One gateway
pm-clearing-cli gateways --gateway TRADER01

# Machine-readable output
pm-clearing-cli --format json gateways
```

Output fields: `gateway_id`, `realized_pnl_total`, `unrealized_pnl_total`, `total_pnl`, `net_qty_total`

Totals are in display currency

The P&L totals in the `gateway_pnl_totals` view are normalized to display currency *per symbol* before being summed, because one tick is worth a different amount at different `tick_decimals` (100 ticks is 1.00 at 2 decimals but 0.01 at 4). This lets a gateway's totals combine symbols with different precisions correctly. The same applies to the per-date totals in `dates --with-totals`. Quantity columns (`net_qty_total`, `traded_qty_total`) are raw share counts and are not scaled.

14.11.3 positions — current positions by gateway and symbol

Returns the full live position state for every `(gateway, symbol)` key. A clearing house uses this to **drill into the detail behind a gateway's aggregate P&L**: it shows exact quantities, average cost, mark price, and the breakdown between buy-side and sell-side notional, which is the starting point for any margin or exposure calculation.

Option	Type	Default	Description
`--gateway GW_ID`	string	all	Filter to one gateway
`--symbol SYMBOL`	string	all	Filter to one symbol
`--limit N`	integer	10 000	Maximum rows returned

```
# All positions
pm-clearing-cli positions

# One gateway, all symbols
pm-clearing-cli positions --gateway MM01

# One symbol across all gateways
pm-clearing-cli positions --symbol AAPL

# CSV export
pm-clearing-cli --format csv positions > positions.csv
```

Output fields include `net_qty`, `avg_cost`, `mark_price`, `tick_decimals`, `realized_pnl`, `unrealized_pnl`, buy/sell quantities and notionals.

Price-derived fields in CLI output — including `avg_cost`, which renders in the same display units as `mark_price` — are normalized using each row's `tick_decimals` (table, JSON, and CSV formats). Use `--raw-output` to see the underlying tick-unit values instead.

14.11.4 pnl — realized/unrealized/total P&L

Returns a focused P&L view without the quantity and notional detail of `positions`. A clearing house uses this as a **daily P&L report**: compliance and risk officers can see at a glance which gateways or symbols are profitable, which are running losses, and whether unrealized exposure is within acceptable bounds.

Option	Type	Default	Description
`--gateway GW_ID`	string	all	Filter to one gateway
`--symbol SYMBOL`	string	all	Filter to one symbol
`--limit N`	integer	10 000	Maximum rows returned

```
# Exchange-wide P&L
pm-clearing-cli pnl

# One gateway
pm-clearing-cli pnl --gateway TRADER01

# One symbol across all gateways
pm-clearing-cli pnl --symbol MSFT
```

14.11.5 daily — daily rollup summaries

Returns per-day aggregates for every `(gateway, symbol)` key, including end-of-day position snapshots. A clearing house uses this for **day-end reconciliation and settlement records**: it provides the official closing quantities, average costs, and P&L figures that feed downstream reporting, fee calculation, and regulatory filings.

Option	Type	Default	Description
`--gateway GW_ID`	string	all	Filter to one gateway
`--symbol SYMBOL`	string	all	Filter to one symbol
`--date YYYY-MM-DD`	date	—	Exact date match
`--from YYYY-MM-DD`	date	—	Inclusive start date
`--to YYYY-MM-DD`	date	—	Inclusive end date
`--limit N`	integer	1 000	Maximum rows returned

```
# Today's summary for all gateways
pm-clearing-cli daily --date 2026-07-05

# A date range
pm-clearing-cli daily --from 2026-07-01 --to 2026-07-05

# One gateway's week
pm-clearing-cli daily --gateway MM01 --from 2026-07-01 --to 2026-07-05

# JSON export for a spreadsheet
pm-clearing-cli --format json daily --date 2026-07-05 > daily_2026-07-05.json
```

14.11.6 trades — raw trade events

Returns the append-only trade fact log exactly as captured from the engine. A clearing house uses this as the **primary audit and investigation tool**: it provides an immutable, timestamped record of every matched fill, which is the source of truth for dispute resolution, trade reconstruction, and regulatory trade reporting.

Option	Type	Default	Description
`--gateway GW_ID`	string	all	Match either the buy-side or sell-side gateway
`--symbol SYMBOL`	string	all	Filter to one symbol
`--date YYYY-MM-DD`	date	—	Exact date match
`--from YYYY-MM-DD`	date	—	Inclusive start date
`--to YYYY-MM-DD`	date	—	Inclusive end date
`--limit N`	integer	200	Maximum rows returned

```
# All trades today
pm-clearing-cli trades --date 2026-07-05

# One symbol, paginated
pm-clearing-cli trades --symbol AAPL --limit 50

# Everything a gateway touched today
pm-clearing-cli trades --gateway TRADER07 --date 2026-07-05

# Date range
pm-clearing-cli trades --from 2026-07-01 --to 2026-07-05 --symbol MSFT
```

The `trades` verb includes `tick_decimals`; `price` is rendered in display units using that precision for all output formats.

14.11.7 exposure — net and gross notional exposure

Returns net and gross notional exposure ranked by size. A clearing house uses this for **real-time risk concentration monitoring**: large gross notional positions signal potential margin pressure, while a high gross-to-net ratio indicates a participant is running offsetting positions that could unwind rapidly.

Option	Type	Default	Description
`--gateway GW_ID`	string	all	Filter to one gateway
`--symbol SYMBOL`	string	all	Filter to one symbol
`--sort FIELD`	string	`gross_notional`	Order rows; see allowed values below
`--limit N`	integer	1 000	Maximum rows returned

```
# Largest exposures first (default sort: gross_notional)
pm-clearing-cli exposure

# Sort by total P&L
pm-clearing-cli exposure --sort total_pnl

# One symbol
pm-clearing-cli exposure --symbol AAPL

# JSON for a risk dashboard
pm-clearing-cli --format json exposure > exposure.json
```

Allowed `--sort` values: `gross_notional`, `net_notional`, `realized_pnl`, `unrealized_pnl`, `total_pnl`

14.11.8 symbols — symbol-level clearing totals

Aggregates all gateways together to show exchange-wide traded volume, notional, and open interest per symbol. A clearing house uses this for **market-level overview and settlement totalling**: it answers which instruments drove the most activity and identifies symbols with large aggregate open positions that may carry overnight risk.

Option	Type	Default	Description
`--date YYYY-MM-DD`	date	—	Restrict daily rollup to an exact date
`--from YYYY-MM-DD`	date	—	Inclusive start date for daily rollup
`--to YYYY-MM-DD`	date	—	Inclusive end date for daily rollup
`--sort FIELD`	string	`symbol`	Order rows; see allowed values below
`--limit N`	integer	1 000	Maximum rows returned

```
# All symbols traded
pm-clearing-cli symbols

# Sort by traded notional (descending)
pm-clearing-cli symbols --sort traded_notional

# Filter to one date
pm-clearing-cli symbols --date 2026-07-05
```

Allowed `--sort` values: `symbol`, `traded_qty`, `traded_notional`, `realized_pnl`, `open_net_qty`

14.11.9 dates — available trading dates

Lists every date for which clearing data exists in the DB. A clearing house uses this as an **index for date-range reporting**: before running a `daily` or `trades` query over a period, use `dates` to confirm which days have data and, with `--with-totals`, to get a quick exchange-wide volume summary per day without querying the full trade log.

Option	Type	Default	Description
`--gateway GW_ID`	string	all	Restrict dates to those where this gateway traded
`--symbol SYMBOL`	string	all	Restrict dates to those where this symbol traded
`--from YYYY-MM-DD`	date	—	Inclusive start date
`--to YYYY-MM-DD`	date	—	Inclusive end date
`--with-totals`	flag	off	Add `traded_qty_total`, `traded_notional_total`, `net_amount_total` columns per date (reads the exchange-wide totals view, so `--gateway`/`--symbol` are ignored)
`--limit N`	integer	1 000	Maximum rows returned

```
# List all dates in the DB
pm-clearing-cli dates

# With per-date volume and net amount
pm-clearing-cli dates --with-totals

# Filter by symbol
pm-clearing-cli dates --symbol AAPL

# Narrow date window with totals
pm-clearing-cli dates --from 2026-07-01 --to 2026-07-05 --with-totals
```

14.11.10 health — process liveness and database health

Checks that `pm-clearing` is alive and actually processing messages. A clearing house uses this as a **liveness and monitoring check**: it detects a stalled or dead clearing process before it causes a gap in positions and P&L. By default two fast checks run:

- **Process** — a `pm-clearing` process is running (found with `pgrep`).
- **ZMQ ping** — `pm-clearing` answers a ping on its health socket (port `5563`) with its current UTC time. The reply comes from the clearing process's own receive loop, so it proves the loop is running, not just that the process exists.

The database check is opt-in with `--extended`, because it runs a full-file `PRAGMA quick_check` that can take many seconds on a large `clearing.db`. It checks that the database passes the integrity check, contains the `trade_events`, `gateway_symbol_positions` and `gateway_daily_summary` tables with the current trade-identity schema, and accepts a (rolled-back) write. The file is never created if it is missing. The check is allowed up to 45 seconds before it is reported as failed.

Option	Type	Default	Description
`--extended`	flag	off	Also check `clearing.db` integrity, schema and read/write access (up to 45 s)
`-v`, `--verbose`	flag	off	List exactly which checks ran; with `--extended`, also print the database size, row counts, last trade and last flush time, and journal mode
`-q`, `--quiet`	flag	off	Print nothing; report only through the exit code

```
pm-clearing-cli health
pm-clearing-cli health --extended
pm-clearing-cli health --extended --verbose
pm-clearing-cli --format json health
pm-clearing-cli health --quiet
```

```
pm-clearing process: OK - pm-clearing is running
clearing database:   OK - skipped (pass --extended to check integrity and read/write access)
clearing ZMQ ping:   OK - replied at 2026-10-17T14:23:37.797827+00:00
health: OK
```

With `--extended --verbose` the report ends with the checks performed and the database metadata — the row counts and freshness figures earlier versions of `health` printed as a table row:

```
Checks performed:
  pm-clearing process (pgrep):   yes
  ZMQ health ping:                yes
  Database integrity/read-write: yes

Database stats:
  path: data/clearing.db
  size: 18.4MB
  trade_events rows: 48211
  gateway_symbol_positions rows: 312
  gateway_daily_summary rows: 1904
  last trade: 2026-07-08T15:59:58.412000+00:00
  last flush: 2026-07-08T16:00:02.977000+00:00
  journal mode: wal
```

The exit code is `0` when healthy and `1` otherwise. With `--quiet` nothing is printed and the exit code is `0` when healthy or `-1` (255 in a shell) otherwise. `--format json` prints the same report as a JSON object (`csv` prints the same text as `table`); `--raw-output` does not apply.

Unlike the query verbs, `health` runs even when `clearing.db` does not exist; a missing database is only reported as a failure under `--extended`.

The health socket binds to `127.0.0.1:5563` by default; set `EDUMATCHER_CLEARING_BIND_HOST` / `EDUMATCHER_CLEARING_` to change where `pm-clearing` binds it, and `EDUMATCHER_ENGINE_HOST` / `EDUMATCHER_CLEARING_HEALTH_PORT` to change where `pm-clearing-cli` connects.

14.11.11 reconcile — verify aggregate consistency (buy and sell sides)

Cross-checks raw `trade_events` fact totals against the pre-computed aggregates in `gateway_daily_summary` for both the buy side and the sell side. A clearing house uses this as a **data integrity gate**: it should return zero rows on a healthy DB, and any discrepancy identifies exactly which date, gateway, symbol, and direction has diverged — pointing directly to the rows that need investigation.

The comparison is a **full outer** one: it is driven by the union of keys on *both* sides, so a `(date, gateway, symbol)` key that exists only in the summaries — the shape produced when every raw row for a key is lost — is reported rather than being invisible. Because `prune` removes raw rows while keeping summaries, pass `--retention-days` (matching the `pm-clearing` retention window) to ignore dates older than the raw-retention window and avoid false positives for legitimately pruned days.

Option	Type	Default	Description
`--gateway GW_ID`	string	both sides	Restrict to one gateway on either buy or sell side
`--symbol SYMBOL`	string	all	Restrict to one symbol
`--from YYYY-MM-DD`	date	—	Inclusive start date
`--to YYYY-MM-DD`	date	—	Inclusive end date
`--retention-days N`	integer	off	Ignore dates older than N days (match `pm-clearing --retention-days`) so pruned-raw days are not flagged

Output columns: `side` (`BUY` or `SELL`), `trade_date`, `gateway_id`, `symbol`, `raw_qty`, `summary_qty`, `qty_diff`, `raw_notiona` `summary_notional`, `notional_diff`.

```
# Full reconciliation (both sides)
pm-clearing-cli reconcile

# Scope to one gateway or date range
pm-clearing-cli reconcile --gateway TRADER01
pm-clearing-cli reconcile --from 2026-07-01 --to 2026-07-05

# Ignore days older than the raw-retention window (avoids false positives
# for pruned days once trade_events has been pruned but summaries remain)
pm-clearing-cli reconcile --retention-days 90

# JSON output (on a clean DB the line "OK — no discrepancies found." is printed
# on stdout ahead of the payload, so filter it or check the exit status)
pm-clearing-cli --format json reconcile
```

14.11.12 sessions — gateway connection and disconnection history

Returns the timeline of every gateway connection recorded by `pm-clearing` from the engine's `system.gateway_auth` (accepted connect) and `system.gateway_bye` (disconnect) broadcasts. A clearing house uses this for **participant access auditing**: it establishes exactly when each participant was active, how long they were connected, and whether a disconnect was clean or forced — context that is essential when investigating missing trades or unexpected position changes.

Option	Type	Default	Description
`--gateway GW_ID`	string	all	Filter to one gateway
`--from YYYY-MM-DD`	date	—	Inclusive start date (matched on `connect_date`)
`--to YYYY-MM-DD`	date	—	Inclusive end date (matched on `connect_date`)
`--connected-only`	flag	off	Return only sessions where `disconnected_at_ns` is NULL
`--limit N`	integer	500	Maximum rows returned

```
# All sessions today
pm-clearing-cli sessions --from 2026-07-05

# History for one gateway
pm-clearing-cli sessions --gateway TRADER07

# Only currently-open sessions (no recorded disconnect)
pm-clearing-cli sessions --connected-only

# Date range
pm-clearing-cli sessions --from 2026-07-01 --to 2026-07-05

# JSON export
pm-clearing-cli --format json sessions > sessions.json
```

Output fields: `gateway_id`, `connect_date`, `connected_at_ns`, `disconnected_at_ns`, `disconnect_reason`

Tip

If a gateway has trades in `trade_events` but no row in `gateway_sessions`, `pm-clearing` was not running when the gateway's `system.gateway_auth` broadcast was published. Restart `pm-clearing` to start capturing session history going forward.

14.11.13 eod — end-of-day sentinel events

Returns the end-of-day sentinel rows written to `session_events` when `pm-clearing` receives `system.eod` from a graceful engine shutdown. A clearing house uses this to **confirm official session close and validate**

end-of-day marks: it answers exactly when the day closed, which mark prices were applied to all open positions at that moment, and provides the authoritative reference point for overnight P&L, margin calls, and daily settlement statements.

Option	Type	Default	Description
`--from YYYY-MM-DD`	date	—	Inclusive start date
`--to YYYY-MM-DD`	date	—	Inclusive end date
`--limit N`	integer	100	Maximum rows returned

```
# Latest EOD markers
pm-clearing-cli eod

# Date range
pm-clearing-cli eod --from 2026-07-01 --to 2026-07-05

# Most recent session close
pm-clearing-cli eod --limit 1
```

Output fields: `id`, `event_type` (always `EOD`), `ts_ns`, `trade_date`, `payload_json` (JSON object with `eod_marks` dict and `symbols_count`).

To extract the EOD mark prices for AAPL from the JSON payload:

```
pm-clearing-cli --format json eod --limit 1 \
  | python3 -c "
import json, sys
row = json.load(sys.stdin)[0]
marks = json.loads(row['payload_json'])['eod_marks']
print('AAPL EOD mark:', marks.get('AAPL', 'n/a'))
"
```

Note

`EOD` rows are only written when `pm-clearing` receives `system.eod` from a graceful engine shutdown. If the engine was killed hard (e.g. `kill -9`), no EOD row is written and the positions retain the last intraday mark.

14.11.14 snapshots — positions as they stood at each close

Returns the rows of `position_close_snapshot`: every position at the moment the session closed, valued at the closing price. This is the data the end-of-day settlement is calculated from (see Closing the Day), so it is the place to look when someone asks "which closing positions did the settlement use?". Values are normalised to display currency unless `--raw-output` is given.

Option	Type	Default	Description
`--gateway GW_ID`	string	—	One gateway
`--symbol SYMBOL`	string	—	One symbol
`--date YYYY-MM-DD`	date	—	One trading date

Option	Type	Default	Description
`--from` / `--to YYYY-MM-DD`	date	—	Inclusive range
`--limit N`	integer	10000	Maximum rows returned

```
# All closing positions of the last day
pm-clearing-cli snapshots --date 2026-11-03

# One trader's closing positions over time
pm-clearing-cli snapshots --gateway GW_BOB --from 2026-11-01
```

Output fields: `trade_date`, `gateway_id`, `symbol`, `net_qty`, `mark_price`, `tick_decimals`, `realized_pnl`, `unrealized_pnl` (cumulative since the start).

14.11.15 prune — remove old raw trade events

Deletes `trade_events` rows older than the given number of days. It does not run `VACUUM`, so the file does not shrink until you run `sqlite3 clearing.db VACUUM` yourself. A clearing house uses this for **storage lifecycle management**: raw fill records accumulate indefinitely; periodic pruning keeps the DB file at a manageable size while preserving all aggregate tables (`gateway_daily_summary`, `gateway_symbol_positions`) that hold the long-running reporting value beyond the raw-event window.

Warning

`prune` is the only write-access verb. It opens `clearing.db` in read/write mode and runs a `DELETE`. It does not need `pm-clearing` to be stopped first: SQLite WAL mode lets readers run alongside a writer, and the prune serializes with `pm-clearing`'s own flushes, but for safety prefer running it during a quiet period.

Option	Type	Default	Description
`--days N`	integer	90	Delete rows older than N days; minimum 1
`--dry-run`	flag	off	Report how many rows would be deleted without deleting any

```
# Delete rows older than 90 days
pm-clearing-cli prune

# Custom retention window
pm-clearing-cli prune --days 180

# See what would be deleted without deleting
pm-clearing-cli prune --dry-run
```

Tip

`pm-clearing` also prunes automatically on startup and on each `system.eod`, so manual pruning is rarely needed. Use `--dry-run` first if you are uncertain.

14.12 Clearing cookbook — common questions and answers

14.12.1 "What is our live P&L right now?"

```
pm-clearing-cli pnl
```

Or for a specific gateway:

```
pm-clearing-cli pnl --gateway TRADER01
```

14.12.2 "Which gateway/symbol positions have the largest open exposure?"

```
pm-clearing-cli exposure --sort gross_notional --limit 10
```

`exposure` returns one row per (gateway, symbol); use `pm-clearing-cli gateways` for per-gateway P&L totals.

14.12.3 "Show me every trade TRADER07 did today"

```
pm-clearing-cli trades --gateway TRADER07 --date 2026-07-05
```

14.12.4 "What was the total exchange volume and notional today?"

```
pm-clearing-cli dates --from 2026-07-05 --to 2026-07-05 --with-totals
```

(`daily --date` returns one row per gateway and symbol, not exchange totals.)

Or for a date range, including net amounts:

```
pm-clearing-cli dates --from 2026-07-01 --to 2026-07-05 --with-totals
```

14.12.5 "Which symbols generated the biggest P&L swings?"

```
pm-clearing-cli symbols --date 2026-07-05 --sort realized_pnl
```

14.12.6 "Did clearing stop updating? When was the last flush?"

```
pm-clearing-cli health
pm-clearing-cli health --extended --verbose
```

The first command tells you whether `pm-clearing` is running and its receive loop still answers. The second also shows `last flush`: if it is more than a few minutes stale during active trading, `pm-clearing` may be alive but no longer receiving trades — check that the engine is up.

14.12.7 "What was the official EOD mark for AAPL yesterday?"

```
pm-clearing-cli eod --from 2026-07-04 --to 2026-07-04
```

This returns the sentinel row written by `pm-clearing` on `system.eod`, including the exact close timestamp and the `payload_json` field which contains the EOD mark prices applied to each symbol.

14.12.8 "When did TRADER07 connect and disconnect today?"

```
pm-clearing-cli sessions --gateway TRADER07 --from 2026-07-05
```

14.12.9 "Which gateways were connected during today's session?"

```
pm-clearing-cli sessions --from 2026-07-05
```

14.12.10 "Which gateways are currently connected (no recorded disconnect)?"

```
pm-clearing-cli sessions --connected-only
```

14.12.11 "Why does TRADER07 have no trades today — was it ever connected?"

Run the session history first:

```
pm-clearing-cli sessions --gateway TRADER07 --from 2026-07-05
```

This tells you whether the gateway connected at all. If no rows are returned, TRADER07 never authenticated. If a row is present with a rapid `disconnected_at_ns`, the gateway was kicked mid-session. Then check trades:

```
pm-clearing-cli trades --gateway TRADER07 --date 2026-07-05
```

14.12.12 "Export today's daily summary for the finance team"

```
pm-clearing-cli --format csv daily --date 2026-07-05 > clearing_2026-07-05.csv
```

14.12.13 "Verify the database is internally consistent"

```
pm-clearing-cli reconcile
# OK — no discrepancies found.
# No rows found.
```

If the reconcile output shows rows, contact the EduMatcher operations team.

14.12.14 "How do I isolate per-session data?"

Set `EDUMATCHER_DATA_DIR` before starting:

```
export EDUMATCHER_DATA_DIR="$HOME/sessions/morning"
pm-clearing &
pm-clearing-cli pnl  # reads morning/clearing.db
```

14.13 Notes

- P&L figures are **gross of transaction costs** — no fees or commissions are deducted.
- Bilateral netting (netting across multiple gateways controlled by the same participant) is not supported; P&L is tracked per individual gateway ID.
- Settlement date tracking (T+1/T+2) is not implemented; all positions are marked intraday.
- `mark_price` is the most recent `trade.executed` price for that symbol during a trading session. On graceful engine shutdown, `pm-clearing` updates `mark_price` to the official EOD `last_price` from the `system.eod` book snapshot (converted to ticks).
- Duplicate trade deliveries (retransmission or replay) are de-duplicated on `(id, ts_ns)` before they reach the ledger, so a replayed trade never double-counts a position; the archive and positions always agree.

- `pm-clearing` raises durable integrity alarms in `session_events`: a `GAP` row when the engine trade-id sequence jumps forward (the PUB feed dropped trades — they cannot be recovered without a replay feed, but the loss is no longer silent). A re-sent trade id is ignored by the archive without an alarm.
- `session_events` and `gateway_sessions` are populated only if `pm-clearing` is running at the time the corresponding messages arrive. A hard engine kill produces no `EOD` row. Gateway sessions whose `system.gateway_auth` broadcast was published before `pm-clearing` was launched will not have a connect row.

14.14 Quick-reference: P&L formulas

Metric	Formula	When computed
Avg cost (adding to position)	$\frac{\text{old_avg} \times \lvert\text{old_qty}\rvert + \text{price} \times \text{qty}}{\lvert\text{new_qty}\rvert}$	Fill increases same-side position
Realized P&L (long closing)	$(\text{fill_price} - \text{avg_cost}) \times \text{close_qty}$	Sell reduces a long position
Realized P&L (short closing)	$(\text{avg_cost} - \text{fill_price}) \times \text{close_qty}$	Buy reduces a short position
Unrealized P&L	$\text{net_qty} \times (\text{mark_price} - \text{avg_cost})$	After every trade event
Total P&L	realized + unrealized	Always

14.15 See also

In this book:

- Order Types — which order types produce fills

Operator's Guide:

- Statistics and Reporting — `pm-stats-cli` for market data queries
- Audit Trail — `pm-audit-cli` for querying the full event log

Reference Manual:

- Processes — how `pm-clearing` connects to the engine

Protocols and Clients:

- Messages — `trade.executed` fields consumed by `pm-clearing`

CHAPTER 15

Your End-of-Day Statement

Learning objectives

After reading this page you will be able to:

- Read your daily statement from top to bottom and explain every line
- Say, in your own words, what *position*, *closing price*, *profit and loss*, *settlement* and *balance* mean
- Work out for yourself whether your statement is right
- Know what to do if you think it is wrong

No finance background is needed. Every word is explained here.

15.1 What the statement is

When the market closes for the day, the exchange adds up how each trader did and writes you a short **statement**, like the one your bank sends you. It tells you three things:

1. **How much you gained or lost today** — and so whether *you owe the exchange* or *the exchange owes you*.
2. **What you still hold** when the market closed.
3. **Every trade you made** that day.

The money is **fictional**: your instructor decides what the numbers mean (points, pretend dollars, a class competition). Nothing real is paid.

Your instructor gives you the statement as a file (usually `statements/<your gateway name>.md`) or by email.

15.2 An example, line by line

Here is the statement of a trader called Bob for Tuesday 3 November 2026:

```
# Statement for 2026-11-03

Bob Berg (GW_BOB) <bob@example.org>

**You owe the exchange 20.00.**

- Today's result: -20.00
- Balance after today: -20.00

## Result by symbol

| Symbol | Result |
|---|---:|
| AAPL | -20.00 |

## Open positions at the close

| Symbol | Quantity | Closing price |
|---|---:|---:|
| AAPL | -150 | 10.20 |

## Trades

| Time (UTC) | Symbol | Side | Quantity | Price |
|---|---|---|---:|---:|
| 10:30:01 | AAPL | SELL | 50 | 10.20 |
```

15.2.1 The heading and your name

`Statement for 2026-11-03` is the **trading date** — the day this statement is about. `GW_BOB` is your **gateway**, the name your trading program connects with. If your name is not listed, the statement shows only the gateway name and adds *"This gateway is not in the roster"* — tell your instructor.

15.2.2 The headline sentence

There are exactly three possible sentences:

It says	It means
The exchange owes you 30.00.	You gained 30.00 today. Think of it as 30.00 *added* to your account.
You owe the exchange 30.00.	You lost 30.00 today. 30.00 is *taken* from your account.
Nothing to pay or receive today.	Your gain for the day was exactly zero — which is also what you get if you took no part in the market and held nothing.

15.2.3 Today's result and your balance

- **Today's result** is the same number as the headline, with a sign: positive means you gained, negative means you lost.
- **Balance after today** is the running total of all your daily results since the start, plus any starting amount your instructor gave you (your *opening balance*). It can be negative; a negative balance simply means you are down overall. It does not stop you trading.

15.2.4 Result by symbol

A **symbol** is the short code of one company's shares (`AAPL`). If you hold or traded several, this table shows how much of today's result came from each. A symbol you no longer hold, did not trade today and made nothing on today is not listed. Each line is rounded to two decimals on its own, so the lines can add up to 0.01 more or less than *Today's result*; the result is the number that counts.

15.2.5 Open positions at the close

Your **position** is how many shares you hold in a symbol.

- A **positive** number means you *own* that many shares (you are **long**). You gain when the price rises.
- A **negative** number means you have *sold shares you did not own* (you are **short**). You gain when the price **falls**. Bob's `-150` means he is short
  150.
- If a symbol is missing, you hold none of it.

The **closing price** is the price of the *last trade of the day* in that symbol. It is the price the exchange uses to put a value on what you hold overnight (this is called **marking to market**).

15.2.6 Trades

Each line is one trade you made: the time (in **UTC**, a worldwide reference time that may differ from your local clock), the symbol, whether you bought (**BUY**) or sold (**SELL**) — or **SELF** if both sides of the trade were yours — how many shares, and at what price.

15.3 Where today's number comes from

Your result is **the change in the value of your account since the previous trading day's close** (on a Monday, that is usually Friday's). Two kinds of profit go into it:

- **Unrealized profit or loss** — on paper, for shares you still hold. If you bought 100 shares at 10.00 and the closing price is 10.50, you are $100 \times 0.50 =$ **50.00 up on paper**.
- **Realized profit or loss** — locked in because you closed part of a position. Buy at 10.00, later sell at 10.30: you realized 0.30 per share.

Add both together and subtract what the total was at the previous close. That difference is today's result.

!!! example "Check Bob's number yourself" On Monday Bob sold 100 shares he did not own at 10.00, and the day closed at 10.00, so he was neither up nor down. Today he sold 50 more at 10.20, so his average sale price is $(100 \times 10.00 + 50 \times 10.20) \div 150 =$ **10.0667**.

```
- At today's closing price of 10.20 he is short 150, so on paper he has
  lost 150 × (10.20 − 10.0667) = **20.00**. (A short position *loses* when
  the price is above what you sold at.)
- At the previous close he was **0.00** down.
- Today he is 20.00 down instead of 0.00 — a change of **−20.00**. That is
  exactly the 20.00 the statement says he owes the exchange.

His balance is −20.00 because this is his first non-zero day. If the price
falls back tomorrow, tomorrow's result is positive and gives some of it
back.
```

Everyone's results add up to exactly zero: whatever one trader gains, another loses. If a statement says you gained, someone else's says they lost the same amount.

15.4 Questions

Why was I paid even though I did not sell anything? Because you hold shares and their price rose. Your holdings are re-valued every day, and you settle the *change*. If the price falls back tomorrow, tomorrow's result is negative and takes it back.

I did nothing today, so why is my result not zero? Because the market moved while you held a position. Doing nothing while holding shares is still a bet on the price.

Does a negative balance mean I am out of the game? No. It has no effect on whether you can trade.

The price on my statement has two decimals — is that all of it? Prices are shown with the precision of the symbol (two decimals for most symbols). Amounts of money always have two decimals.

My statement looks wrong. What now? Tell your instructor which line you doubt. They can re-calculate without changing anything (`pm-eod run --date <date> --dry-run`) and show you the closing positions (`pm-clearing-cli snapshots --date <date>`) and your trades (`pm-clearing-cli trades --gateway <your gateway>`). If a mistake is found, the day is re-done and a corrected statement is written.

15.5 Words used on this page

Word	Meaning
Share	A small slice of ownership in a company
Symbol	The short code for a company's shares, such as `AAPL`
Trade	A buyer and a seller agreeing on a price and a quantity
Gateway	The connection your trading program uses; it has a name such as `GW_BOB`
Position	How many shares you hold (negative = short)
Long / short	Holding shares / having sold shares you do not own
Closing price	The price of the last trade of the day
Marking to market	Valuing what you hold at the latest market price

Word	Meaning
Unrealized P&L	Profit or loss on paper, on shares you still hold
Realized P&L	Profit or loss locked in by closing a position
P&L	"Profit and Loss"
Settlement	Turning each trader's gain or loss for the day into an amount to pay or receive
Balance	Your running total of daily results plus your opening balance
Trading date	The calendar date, in the exchange's own timezone, of the trading day

For the full mechanics see Positions and P&L. How the exchange produces this statement is described in the operator's chapter Closing the Day.

CHAPTER 16

Trader Information Terminal — "TapeDeck" (`pm-terminal`)

Learning objectives

After reading this page you will understand:

- What the Trader Information Terminal is for, and when to use it instead of a trading or administration tool
- Which services must be running before the display becomes useful
- How to start the terminal locally or in a container
- What each of the seven screens is meant to help a viewer understand
- How to recognise normal disconnected, missing-history, and no-index states without mistaking them for broken screens
- Which settings an operator is most likely to adjust

16.1 Overview

TapeDeck is the friendly nickname this guide uses for the Trader Information Terminal, whose system name is `pm-terminal` and whose source code lives in `web-apps/terminal-gui/`. It is a read-only market display for the EduMatcher exchange: a browser window for watching live prices, trades, auctions, halts, indexes, and depth-of-book information.

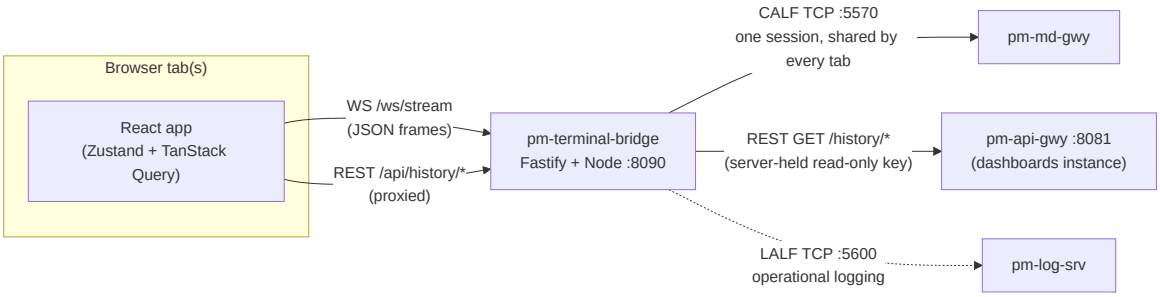
It is deliberately *not* a trading application. There is no order entry, no login screen, and no write path from the browser back into the exchange. Use it for a classroom wallboard, a demo display, an observer workstation, or a quick operator check that the market-data feed is alive. Use the trading client for placing orders, and the administration tools for changing exchange state.

The terminal shows live data from the CALF market-data feed and reads historical bars and index history through the API Gateway. The browser never sees an API key; the small bridge process running beside the web UI holds the upstream connections.

16.1.1 What starts when you run it

Running TapeDeck starts one server process, `pm-terminal-bridge`, and serves the web page from that same process in the container setup. The bridge opens one live CALF connection to `pm-md-gwy`, shares that single feed across every open browser tab, and opens history requests to `pm-api-gwy` when charts or previous-close data are needed. Optional operational logs go to the centralized log server when it is available.

Most users do not need the following diagram to operate the terminal, but it is useful when deciding which host names and ports to put in the container environment.



16.2 Prerequisites

Requirement	Notes
`pm-md-gwy` (CALF gateway)	Required for live prices, trades, session state, auctions, halts, indexes, and depth. The terminal can start without it, but it will show `RECONNECTING`/`OFFLINE` until the feed is reachable.
`pm-api-gwy` (API Gateway) with a read-only API key	Required for charts, previous-close data, index history, and Overview/Movers `Open`/`Volume` columns. Live prices still tick without it.
`pm-log-srv` (Centralized Log Server) — optional	Used only for operational logs. If it is unavailable, the terminal still starts and writes to stdout or the local failover log directory.
Podman **4** or **Docker** **24** with a Compose plugin	Needed for the recommended container run path.
Node.js **20** and **npm** **10**	Needed only for local development without a container.

16.3 Running the application

TapeDeck needs three things from the exchange, and how you supply them is the only real difference between the ways of running it:

It needs	For	Without it
`pm-md-gwy` on 5570	The live CALF feed — prices, depth, trades	The page loads but shows `RECONNECTING`
`pm-api-gwy` on 8081, plus a read-only API key	Charts, previous close, historical index data	Live prices tick; history panels stay empty
`pm-log-srv` on 5600 *(optional)*	Sending the bridge's own logs to the central log server	The bridge writes a fallback log to `./logs`

16.3.1 The whole stack (recommended)

This is the path that requires nothing of you: it starts the exchange and TapeDeck together, resolves the read-only API key out of the deployed configuration, and points the terminal at the backend by service name.

A released install:

```
cd ~/.edumatcher
./edumatcher.sh start
```

From a source checkout:

```
cd deployment/docker
make up-all
```

Then open **http://localhost:8090**. The log console comes up on 8091, the trading GUI on 8093 and the order book viewer on 8094 and the operator dashboard on 8095 at the same time.

There are no addresses to configure because every container shares one Compose network, on which the exchange answers to the hostname `edumatcher`. The API key is the one part that cannot be a fixed default — it is generated per engine configuration — so `up-all` reads it from the deployed configuration and injects it. See Where the read-only API key comes from and Installation.

16.3.2 This app alone, in a container

Use this when the exchange is running somewhere else — another machine, a VM, or as processes on your host. From `web-apps/terminal-gui/`:

```
export EDUMATCHER_DATA_DIR=~/.local/share/edumatcher
make up
```

Then open **http://localhost:8090**. `make logs` follows the bridge log, `make down` stops it.

How `make up` finds the API key. An explicit `PM_TERMINAL_API_KEY` always wins:

```
make up PM_TERMINAL_API_KEY=key-readonly-...
```

Otherwise `make up` reads it from the deployed configuration `$EDUMATCHER_DATA_DIR/ref_data/engine_config.json`, picking the first credential with `gateway_id: null` (the `dashboards` instance is preferred when several carry one). It prints which instance and port the key came from, and warns when `API_GATEWAY_URL` points at a different port. When there is no key to be found it warns and starts anyway — the live feed needs no key, only the history panels do. Pass the key explicitly when the exchange runs on another machine or in a VM, since the look-up reads this host's data directory.

Now the addresses matter, because the container is no longer beside the exchange. The compose file defaults to `host.docker.internal`, which resolves on Docker Desktop but not on Podman or Linux Docker:

```
export CALF_HOST=host.containers.internal
export API_GATEWAY_URL=http://host.containers.internal:8081
export LOG_SRV_HOST=host.containers.internal
make up
```

The history key is valid on 8081, not 8080

`API_GATEWAY_URL` must point at the `dashboards` instance. The read-only credential (`gateway_id: null`) is issued there; `desk` on 8080 will reject it, and the symptom is a live book with empty charts. The compose file already defaults to `:8081` — keep the port when you change the host.
If port 8090 is taken, move the *host* side of it. Pick something outside 8090–8095, which the other applications use:

```
TERMINAL_GUI_PORT=8100 make up
```

The container serves the built React frontend, the WebSocket endpoint, and the small read-only history proxy from the same port. There is no database volume: the only bind mount is `./logs:/app/logs`, used when the optional log server is not reachable after startup.

Alternative: direct Compose commands

```
PM_TERMINAL_API_KEY='...' docker compose up --build -d
docker compose logs -f terminal-gui
docker compose down
```

With Podman, use `podman-compose` for the same commands.

16.3.3 Running on a separate display server

TapeDeck does not have to run on the same machine as the matching engine, `pm-md-gwy`, or `pm-api-gwy`. A common deployment is:

- **Exchange server**: runs the engine, `pm-md-gwy`, `pm-api-gwy`, and optionally `pm-log-srv`.
- **Display server**: runs only the TapeDeck container and serves the browser UI to viewers.

No protocol change is needed on the exchange side. The terminal bridge is just another external CALF client plus a read-only history client. The practical requirements are:

Exchange-side item	What to check
`pm-md-gwy`	It normally binds to `0.0.0.0:5570`, so no application change is needed unless your config deliberately set `market_data_gateway.bind_address` to `127.0.0.1`. The display server must be able to open TCP `5570` on the exchange host.
`pm-api-gwy`	It binds `0.0.0.0` by default, so no application change is needed unless your configuration deliberately narrows `api_gateways.<name>.host`. The display server must reach the instance that issues the **read-only** credential — the `dashboards` instance, HTTP `8081` in the bundled configurations, not `desk` on 8080.

Exchange-side item	What to check
API key	Create or reuse a read-only `pm-api-gwy` key with `gateway_id: null`; TapeDeck only needs history reads and never sends the key to browsers.
`pm-log-srv`	Optional. If you want centralized terminal logs, make TCP `5600` reachable and set `LOG_SRV_HOST` on the display server. If not, set `LOG_SRV_ENABLED=false` or let the container write its fallback log to `./logs`.
Firewall / routing	Open only the ports the display server actually needs: TCP `5570` for CALF, TCP `8081` for history, and optionally TCP `5600` for logs. Browsers only need access to the display server's `8090` port, not to the exchange gateways.

On the display server, use the prepared image rather than building from source. The exact image name depends on how the release was delivered:

```
# Option A: the released image
podman pull ghcr.io/johan162/edumatcher-terminal-gui:<VERSION>

# Option B: an image tarball made with `make cdist` in web-apps/terminal-gui
podman load --input edumatcher-terminal-gui-<VERSION>.tar.xz
```

Then run the container, pointing it at the exchange server's network name or IP address:

```
mkdir -p logs

podman run -d --name terminal-gui \
  --restart unless-stopped \
  -p 8090:8090 \
  -v "$PWD/logs:/app/logs" \
  -e CALF_HOST=exchange.example.org \
  -e CALF_PORT=5570 \
  -e API_GATEWAY_URL=http://exchange.example.org:8081 \
  -e PM_TERMINAL_API_KEY='...' \
  -e INDEX_IDS=MAIN \
  -e LOG_SRV_ENABLED=false \
  ghcr.io/johan162/edumatcher-terminal-gui:<VERSION>
```

Use the image name you pulled or loaded — a loaded tarball is named `edumatcher-terminal-gui:<VERSION>` — and `docker` instead of `podman` if that is your container runtime. If centralized logging is available, replace `LOG_SRV_ENABLED=false` with:

```
-e LOG_SRV_ENABLED=true \
-e LOG_SRV_HOST=exchange.example.org \
-e LOG_SRV_PORT=5600
```

After startup, open **http://display-server.example.org:8090** from a browser. If the page loads but shows `RECONNECTING`, the display server can serve the UI but cannot reach `pm-md-gwy`. If live prices tick but charts or previous close values are missing, check `API_GATEWAY_URL` and `PM_TERMINAL_API_KEY`.

16.3.4 Local development

From the `web-apps/terminal-gui/` directory:

```
make install    # npm workspace install; once, and after a dependency change
make dev        # Vite dev server on :8190, bridge on :5190
```

Open **http://localhost:8190**. You do not talk to the bridge directly: `vite.config.ts` proxies `/api` and `/ws` from the dev server to the bridge on `127.0.0.1:5190`, so the browser sees a single origin.

The bridge's own connect targets all default to `127.0.0.1`, which is exactly where the container stack publishes its ports — so `make up-all` in `deployment/docker` plus `make dev` here works with no configuration beyond the API key. `make dev-env GUI=terminal-gui` in `deployment/docker` prints the two values that cannot be defaulted:

```
eval "$(make -s -C ../../deployment/docker dev-env GUI=terminal-gui)"
make dev
```

Keep dev-env out of the shell you start containers from

`eval "$(make dev-env …)"` exports `API_GATEWAY_URL=http://127.0.0.1:8081` and `PM_TERMINAL_API_KEY` into your shell, and they stay there. That is right for `make dev`, whose bridge runs on your own machine. It is wrong for a **container** started later from the same shell: Compose hands the exported `API_GATEWAY_URL` to the container, where `127.0.0.1` is the container itself. The app's own `make up` in `web-apps/terminal-gui` — or `make up` in `web-apps/` without `VM_BACKEND_IP` — then starts a bridge that cannot reach `pm-api-gwy`: live prices still tick, but the charts, previous close and Open/Volume columns stay empty, and the bridge log shows the history requests to `127.0.0.1:8081` failing. The exported key also takes precedence over `make up`'s own look-up, and is stale once the exchange runs another configuration.
`make up-all` and `./edumatcher.sh start` are not affected: whenever they find the read-only credential they set both values themselves.
Either load the variables for the development server only, in a subshell:

```
(eval "$(make -s -C ../../deployment/docker dev-env GUI=terminal-gui)"; make dev)
```

or clear them before starting a container from that shell:

```
unset API_GATEWAY_URL PM_TERMINAL_API_KEY
```

If `pm-md-gwy` is not running the page still loads, with the connection indicator showing `RECONNECTING`/`OFFLINE` until the feed appears. `make dev-bridge` runs only the bridge, `make dev-web` only the web server, and `make test` the Vitest suite. The full inner-loop workflow is The Development Loop.

16.3.5 Where the read-only API key comes from

You never type the API key in the normal start paths, yet TapeDeck can read history. The key has always been there: it is part of the exchange's own configuration, and every start path looks it up and hands it to the bridge.

1. It is generated with the configuration. `pm-config-gen`'s `--api-gateway-readonly-key` option adds one extra credential to the `dashboards` API gateway instance (port 8081), with `gateway_id: null` and a randomly generated key. Every bundled example configuration was generated with that option (see its `mkrefdata.sh`), so each carries one — and each a different one. In `docs/examples/ref_data/s1-basic-setup/engine_config.yaml`:

```
api_gateways:
  dashboards:
    port: 8081
    credentials:
    - api_key: key-readonly-fq9m76
      gateway_id: null
      description: Generated read-only market-data key
```

When a configuration is deployed, the key travels with it into `<data dir>/ref_data/engine_config.json`, the file every exchange process reads.

2. The start path looks it up. The bridge takes the key from the environment variable `PM_TERMINAL_API_KEY`, and each way of starting TapeDeck fills that variable from the deployed configuration:

Start path	How it finds the key
`make up-all` / `./edumatcher.sh start`	Starts the exchange first and waits for its configuration to be deployed, then reads `/data/ref_data/engine_config.json` inside the exchange container. It exports the key as `PM_TERMINAL_API_KEY` and sets `API_GATEWAY_URL=http://edumatcher:<that instance's port>`, and only then starts the web applications — which is why the whole-stack start runs in two phases
`make up` in `web-apps/terminal-gui`	Reads `$EDUMATCHER_DATA_DIR/ref_data/engine_config.json` on this host. A `PM_TERMINAL_API_KEY` you set yourself always wins
`make dev-env GUI=terminal-gui`	Reads the key from the running exchange container and prints it as an `export` line for `make dev`

All three pick the first enabled gateway instance, in name order, that carries a credential with `gateway_id: null` — so `dashboards` is preferred over `desk` when both have one — and take that instance's port along with the key.

Why this is acceptable. A read-only key is not a secret that grants anything worth protecting: it reads only public data — reference data, the market-data stream and history — and can neither place nor see orders. It still never reaches the browser; only the bridge holds it.

A configuration without one. A configuration you built yourself may have no `gateway_id: null` credential. TapeDeck then still starts, with a warning: live prices work, because they come from `pm-md-gwy` and need no key, but the charts, previous close and Open/Volume columns stay empty. To fix it, add a credential with `gateway_id: null` to the `dashboards` instance — in the Configuration GUI's gateway settings, by hand in `engine_config.yaml`, or by regenerating with `--api-gateway-readonly-key` — and redeploy.

Switching configuration changes the key

Because every configuration has its own key, a key copied from one configuration is rejected (HTTP 401) by an exchange running another. The automatic look-up avoids this; a key you exported yourself, or one

left in your shell by `make dev-env`, does not.

16.4 A tour of the interface

Page 1 of 20 on a 300-symbol exchange, advancing every 8 seconds. AAPL and ADBE are starred into the watchlist (the toggle counts five pinned symbols), ADBE carries a `HALT` badge, and the status strip reports `2 symbols halted` and the age of the last tick beside the CALF state. Every `%CHG` has its `*`: this exchange has no previous close on record yet, so change is measured from today's open.

16.4.1 Top bar

A single row (not a collapsible sidebar — seven destinations is small enough for one row, and a data-dense terminal wants its horizontal space for numbers, not navigation chrome) holding:

- **`EduMatcher pm-terminal v<version>`** — the app name and the release it was built from.
- The seven view tabs: **Overview**, **Symbol**, **Index**, **Tape**, **Movers**, **Session**, **News**.
- A **density** control (gauge icon) that cycles **Lobby → Standard → Dense**. This is a display preference, not a mode — every route and every data point stays reachable at every setting; only defaults change (larger type and a longer page delay under Lobby, tighter rows and shorter delay under Dense). It persists to the browser's `localStorage`, so a different browser or profile always starts on **Standard**.
- A **theme** toggle (dark by default — the working default for a trading screen — with a full light palette for bright rooms and projectors).
- A **settings** cog holding **Font size** — XS, S, M, L, XL or XXL, XS by default — which scales the whole page, for a projector or a screen read from across a room. It is independent of density, persists to the browser's `localStorage`, and (as the popover says) works in Chrome and Safari only.
- A **connection indicator**: `LIVE` (green), `RECONNECTING` (amber), or `OFFLINE` (red), reflecting the bridge's own CALF session state, plus the gateway id it is talking to.

16.4.2 Status strip (footer)

A single-line summary meant to be readable from across a room. It opens with the session indicator every EduMatcher GUI shows, exactly as the Trading GUI does in its top bar: the session phase as a coloured badge, the next phase and the time left to it when the feed has named one (`Continuous → Closing Auction in 03:08`), and the clock in the exchange session timezone. The badge reads `Unknown` until the feed has reported the phase, and whenever the CALF connection is not live. Then the number of currently-halted symbols, the total symbol count, the CALF connection state and **the age of the last market-data tick**.

Connection state and data age are two different readings, on purpose

"CALF connected" only says the pipe is open — it says nothing about whether anything is actually coming down it, and a feed that has gone silent behind a healthy socket is exactly the failure a reader most needs to catch. The status strip shows both: connection state, and separately, how long it has been since the last tick arrived. A silently frozen exchange still reads "CALF connected" but its tick age keeps climbing.



Figure 16.1: The Market Overview: the top bar, the first page of the symbol grid and the status strip.

16.5 Screen tour

16.5.1 Market Overview

The default landing view: every tradable symbol, paginated, meant to run unattended on a classroom or lobby display just as well as be actively browsed.

Key behaviors:

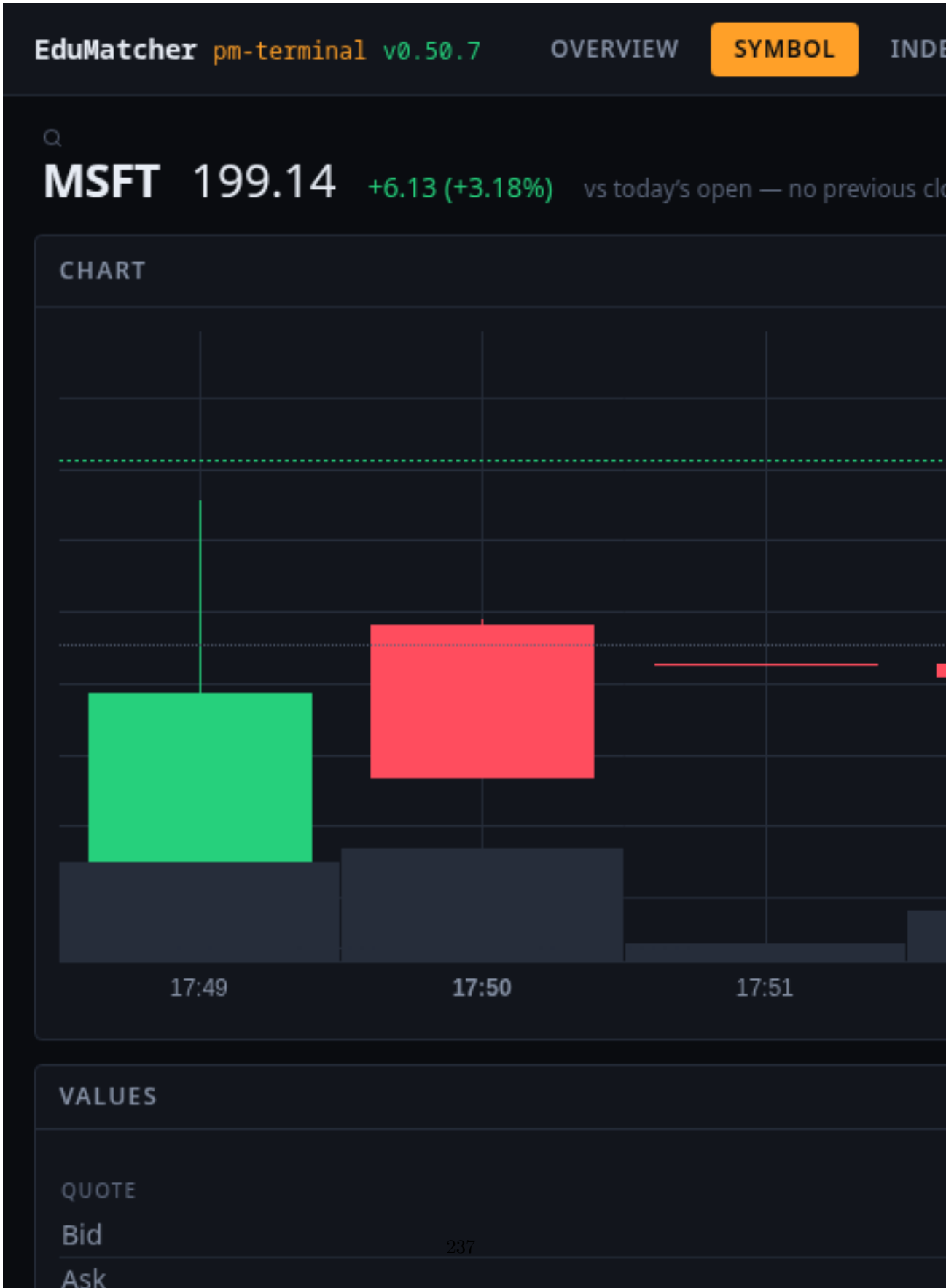
- **Auto-paging.** Rows are split into pages sized to fit the viewport (⚙ control offers 3s/5s/8s/15s/30s dwell times, or a density-based default), so the grid never needs to scroll — useful for an unattended display with no mouse. Hovering the grid, sorting a column, or typing in the symbol search **suspends** auto-advance (a reader who is actively interacting with the board should not have it slide out from under them); the manual ‹/›/pause controls keep working regardless.
- **Every row stays live on every page.** Paging is purely a rendering concern — the bridge already holds one wildcard subscription covering every symbol, so there is no per-page subscribe/unsubscribe to do, and numbers on a page you are not currently viewing never go stale.
- **Sortable columns** and a **type-ahead symbol search** — both narrow/order what is shown without touching the underlying subscription.
- **Watchlist.** Click the ☆ next to any symbol to pin it; the `All`/☆ `Watchlist` toggle switches the grid between paging through every symbol and paging through only pinned ones. This is client-only, `localStorage`-persisted state — there is no server-side watchlist and nothing to log into.
- **A row fades after a configurable silence threshold** (`fade …` control — choices from 1 minute to 1 hour, or off). The right value is a property of the exchange, not of the terminal: a busy, liquid book and a thin classroom exchange want very different thresholds, so it is exposed rather than hardcoded.
- **During a call auction phase**, the quote columns (`Bid/Ask`) are replaced by auction-indicative columns (indicative uncross price/quantity, imbalance) for symbols currently in an opening or closing auction — a call phase is a different kind of market, not a display preference, so the grid follows it automatically rather than offering a toggle.
- **Not-executable banners.** When the whole board is outside continuous trading (closed, or in a call auction), a banner says so explicitly rather than merely dimming the numbers — the prices and volumes shown remain an accurate record, they are just not currently tradable. A separate banner appears if the previous-close lookup failed (percentage change is then measured from today's open instead, and marked with a small *) or if the history service itself is unreachable (Open/Volume/Turnover columns go blank; live prices are unaffected).

16.5.2 Symbol Detail

The deep-dive view for one instrument: a candlestick + midpoint chart, a values table, and an optional depth ladder. Large-screen only, by design — there is no responsive mobile layout.

One-minute OHLC candles with their volume, the VWAP line and the last price (Midpoint is toggled off). Below, the Values panel and, beside it, the Depth-of-Book ladder: the best prices meet in the middle, the # column counts the orders at each level, and the percentage beside each price is its distance from the touch, which shows at a glance that the levels around 192.9 and 204–207 are about 3% away.

- **Header.** Symbol, session badge, last price, change and %change (always quoted against the *previous close*, footnoted when no previous close is on record and the figure falls back to today's open instead), and today's volume.
- **Chart.** Time-window presets (`1D/5D/1M/3M/YTD/All/Live`), free-form drag-zoom, and two indepen-


EduMatcher pm-terminal v0.50.7
OVERVIEW
SYMBOL
INDE
MSFT 199.14 +6.13 (+3.18%) vs today’s open — no previous cl
CHART
17:49
17:50
17:51
VALUES
QUOTE
Bid
Ask

dently toggleable series: OHLC candlesticks (built from historical bars, with the live-forming bar updated in place from CALF `TRADE` prints) and a spliced midpoint line — a coarser, 15-minute-resolution historical segment giving way to a tick-by-tick live segment from CALF `TOP`, drawn in a slightly muted style where it is the coarser data. A reference line for the previous close is always drawn, and a VWAP line appears on the `1D/Live` presets only (VWAP is a same-session benchmark; drawing it across a multi-day preset would be quoting today's average against days it has nothing to do with).

- **Auction and halt context.** An auction uncross fills a dismissible banner (equilibrium price or "no cross," matched quantity, and any residual imbalance) that distinguishes an opening/closing auction, a circuit-breaker **reopening** auction, and a startup/recovery uncross by name, rather than calling all three "auction uncrossed." A halted symbol expands to show the circuit-breaker detail — trigger level, trigger and reference price, and a **corridor bar** showing the price band the symbol may reopen inside, with the last indicative price marked against it. This is the visual explanation of *why* a halt is still running: if the marker sits outside the band, the call phase was extended rather than printing.
- **Values panel and Depth ladder.** The Values panel is always shown; the ☐ `Depth` toggle fills the panel beside it with the Depth-of-Book ladder, which otherwise only says to turn the toggle on. It is off by default on purpose: unlike `OHLC/Midpoint`, which reuse subscriptions the bridge already holds for every symbol, turning Depth on causes the bridge to open a brand-new per-symbol CALF subscription — so it is opt-in per viewer, a heavier, deliberately-requested data stream.

16.5.3 Depth-of-Book

A Level 2 ladder (aggregated quantity per price level — never per-order identity, which CALF does not carry at any version) for whichever symbol currently has the Depth toggle on in Symbol Detail. It is not its own navigation tab.

Columns run outward from the touch in both directions — `Cum | Qty | # | Bid ‖ Ask | # | Qty | Cum` — so the two best prices meet in the middle and the cumulative totals sit at the outer edges, where the eye ends up after scanning inward. Two things worth calling out:

- **The # column** is the count of individual resting orders aggregated into that price level — genuinely useful context a bare quantity doesn't convey (1,400 shares from one order reads very differently than the same 1,400 split across four), and it costs nothing extra since `COUNT` is already on the CALF wire.
- **Rows are evenly spaced regardless of price gaps**, with the actual distance from the touch shown as a percentage figure beside each row instead. Spacing rows by price would collapse to unreadable slivers the moment one level sat far out; the trade-off is that a lone level far from a tight cluster would otherwise look like just the next rung down, which is exactly what the percentage-distance figure (and a subtle highlight when a level is unusually far out) corrects for.

A **Bid depth / Ask depth / Imbalance** summary line below the ladder states the book's lean as a percentage (e.g. "62% bid") rather than a bid:ask ratio, since a ratio's useful range is lopsided (0.2 and 5.0 are the same imbalance mirrored) while a percentage reads the same distance from 50% either way.

16.5.4 Index View

Headline level and a historical chart for a configured exchange index (see Market Index).

The headline level, change, and session badge always come from the **live** CALF `INDEX` stream, never from a historical REST row for the current date — `/history/index-daily`'s `close_level` is only guaranteed final once the session for that date has actually closed, so quoting it live for *today* would risk showing a figure that is still moving. Open/High/Low are safe to read from the REST row even intraday, since those are

running-so-far values that only get more accurate as the day progresses. Switching indexes (when more than one is configured) costs nothing upstream — the bridge holds a standing subscription for every configured index regardless of which one a given tab is currently viewing. If the exchange has no index configured at all, the tab still exists and shows an explicit empty state rather than being hidden, so the tab row never shifts between differently-configured classroom exchanges.

16.5.5 Trade Tape / Time & Sales

Every print on the exchange, newest first, filterable by symbol.

The symbol filter narrows what is *displayed*; the bridge's underlying `SYM=*` wildcard subscription means every symbol's prints are already arriving regardless of the filter, so switching it is instant and free. `Pause` freezes the visible rows without losing anything — the tape keeps recording underneath, and `Resume` shows the current state rather than a gap where the pause was.

Gap markers are a real, shipped feature, not a hypothetical

If the bridge's CALF connection drops and cannot fully repair a symbol's trade sequence on reconnect (the replay window has already rolled past the missed messages), the tape shows an explicit marker row — *"gap in the tape — some prints for `SYMBOL` were missed"* — in place among the prints it falls between, rather than silently omitting the missing prints or, worse, saying nothing at all. A record with an unmarked hole in it is worse than one that admits the hole, because a viewer has no way to tell it apart from a genuinely quiet stretch. This is the direct result of a real CALF protocol fix made while building TapeDeck.

16.5.6 Movers

A different ranking over the same data the Overview grid already computes — no new subscriptions.

Three tabs: **Gainers** and **Losers** rank by percentage change from the previous close (falling back to today's open, and labelled as such, when no previous close is available); **Active** ranks by session turnover (value traded) instead — a common third view on real market boards, and cheap here since Overview already computes turnover per symbol. Each row's bar is scaled relative to the largest value currently shown on that tab.

16.5.7 Session & Halt Status Board

Three panels in one view: the exchange-wide session phase, every symbol currently halted with its full circuit-breaker detail, and the auctions that have uncrossed since the tab was opened.

Two operator halts (ADBE, AMZN): level `ADMIN_SYMBOL`, no trigger or reference price, resuming only when the operator clears them. Below, the opening auction's results, newest first: the equilibrium price, the matched quantity, the number of trades and the residual imbalance per symbol, and `(no cross)` where nothing could match.

- **Session status** names the current phase and places it in the exchange's timetable, all in the exchange's time zone:
  - *This week*: each day's hours from pre-open to the close, with closed days dashed and bank holidays named.
  - *Timetable*: the day's phases in order (Closed, Pre-Open, Opening Auction, Continuous, Closing Auction, Closed), each arrow labelled with the time of that transition. The current phase is filled in the top bar's colour and marked NOW. When today does not trade, the next trading day is



Figure 16.3: The Session & Halt Status Board just after an opening auction.

shown instead.
  - A bar of the whole day from 00 to 24, with a marker at the exchange's current time. The marker is hidden while the session is off its timetable, for example after an operator moved it by hand, or when `pm-scheduler` runs a compressed simulation clock.
  - *Next bank holiday*: the next holiday that falls on a normal trading day, and whether the exchange closes or runs holiday hours.
- **Active halts** shows, per halted symbol: circuit-breaker level, trigger and reference price (blank for an operator-initiated halt, which carries neither), how and when it resumes (a converted wall-clock time for a timed halt, or `Manual` for one that only ends on an explicit operator action), and how long it has been halted.
- **Recent auction results** is a session-scoped, client-side ring buffer of every auction uncross seen since the tab opened — not a durable audit trail (that is `pm-index`'s own structural log, surfaced separately via the Index View's "Recent changes" strip). An omitted equilibrium price is labelled `(no cross)` rather than shown as a blank, since "no crossable interest at all" is a meaningfully different outcome from a price, just an unusual one.
- Opening this view is itself one of the two triggers for the bridge to hold a per-symbol `CB` subscription (the other is opening that symbol's own Symbol Detail view) — closing the tab releases every subscription it was the sole reason for.

16.5.8 News

The headlines `pm-market-sim` publishes (see The market model), newest first, with the same badges and filter as the trading GUI's News screen: `RUMOUR` with its credibility, `CONFIRMED`/`RETRACTED` when a rumour resolves (a retracted rumour and its retraction struck through), and a filter that takes a symbol or a sector.

CALF carries no news, so this view reads them through the bridge's `/api/news` proxy to `pm-api-gwy`'s `GET /api/v1/news` — the same read-only key as the charts — once a second while it is open. Without `pm-api-gwy` it says the news is unavailable; without `pm-market-sim` the list stays empty.

16.6 Configuration reference

The container is configured with environment variables. Most installations only need to set `PM_TERMINAL_API_KEY` and, when the upstream services are not on the default host names, `CALF_HOST`, `API_GATEWAY_URL`, and `LOG_SRV_HOST`.

Variable	Container default	Purpose
`TERMINAL_GUI_PORT`	`8090`	Host port exposed by `docker-compose.yml`; use this when `8090` is already in use.
`HOST` / `PORT`	`0.0.0.0` / `8090`	Bridge bind address inside the container.
`CORS_ORIGIN`	*	CORS allow-list
`STATIC_DIR`	`/app/apps/web/dist`	Serve a built frontend from here (single-container mode); unset in development
`MAX_WS_CLIENTS`	`200`	Browser-tab cap

Variable	Container default	Purpose
WS_HEARTBEAT_INTERVAL_SEC	5	How often every tab receives the bridge's status heartbeat
WS_PING_INTERVAL_SEC / WS_PING_MAX_MISSED	10 / 2	Ping to each tab, and missed replies before a dead tab is dropped
WS_MAX_BUFFERED_BYTES	5000000	A tab whose outgoing buffer exceeds this (a stalled browser) is disconnected
CALF_HOST / CALF_PORT	host.docker.internal / 5570	pm-md-gwy; use host.containers.internal for Podman if needed.
CALF_CLIENT_ID	pm-terminal-bridge	CALF HELLO.CLIENT
CALF_PING_INTERVAL_SEC	60	Keepalive; belt-and-braces now that the gateway's idle timer honours outbound traffic too
INDEX_IDS	—	Comma-separated index ids to subscribe to (CALF has no "list the indexes" request)
API_GATEWAY_URL	http://host.docker.internal:8081	pm-api-gwy, **the dashboards instance** — the read-only key is not valid on desk (8080). Use host.containers.internal for Podman. In the whole-stack path this is http://edumatcher:8081, set for you.
PM_TERMINAL_API_KEY	—	Read-only (gateway_id: null) key, history reads only — never sent to the browser
LOG_SRV_ENABLED	true	false skips even the startup probe
LOG_SRV_HOST / LOG_SRV_PORT	host.docker.internal / 5600	pm-log-srv; use host.containers.internal for Podman if needed.
LOG_SRV_CLIENT_ID	pm-terminal-bridge	Name the bridge registers with at the log server
LOG_SRV_INSTANCE	—	Optional suffix that tells several terminals' logs apart
LOG_CONNECT_TIMEOUT_SEC	0.5	Startup probe and each reconnect attempt

Variable	Container default	Purpose
`LOG_FAILOVER_TIMEOUT_SEC`	`30`	Grace window before the one-way switch to a local log file
`LOG_QUEUE_MAXSIZE`	`2000`	Bounded backlog while reconnecting
`LOG_FAILOVER_DIR`	`/app/logs`	Where the post-failover log file goes; bind-mounted to `./logs` by Compose.

16.7 Troubleshooting

Symptom	Likely cause	What to check
Every screen shows the red "Disconnected from pm-terminal-bridge" banner	The browser's own WebSocket to the bridge is down	Confirm the bridge process is running and reachable at `HOST:PORT`; check the browser console for the WS connection error
Connection indicator shows `RECONNECTING` and never returns to `LIVE`	The bridge cannot reach `pm-md-gwy`	Confirm `CALF_HOST`/`CALF_PORT` point at a running gateway, and that nothing (e.g. a firewall) blocks that TCP connection from the bridge's host
Charts and Open/Volume columns are empty or show an "unavailable" banner, but live prices still tick	The bridge cannot reach `pm-api-gwy`, or `PM_TERMINAL_API_KEY` is missing/invalid	Check the bridge's startup log for a `PM_TERMINAL_API_KEY is unset` warning; confirm `API_GATEWAY_URL` is correct and the key is a valid read-only history key
A container started with `make up` shows live prices but no history, and its log shows requests to `127.0.0.1:8081` failing	`API_GATEWAY_URL=http://127.0.0.1:8081` was left in the shell by `eval "$(make dev-env …)"`	`unset API_GATEWAY_URL PM_TERMINAL_API_KEY` and run `make up` again — see Local development
Index tab shows "This exchange has no index configured"	Expected, not an error, when no index is configured for this exchange	Set `INDEX_IDS` if an index should be shown
A tape gap marker appears	The bridge's CALF connection dropped for long enough that the gateway's replay window rolled past the missed trades	Expected behavior under a real disconnect — see Trade Tape above; not itself a bug to fix

Symptom	Likely cause	What to check
Prices render to the wrong number of decimal places	The connected `pm-md-gwy` predates the `REF=` per-symbol precision field	Upgrade the gateway; TapeDeck falls back to two decimal places when `REF=` is absent, which is a compatibility fallback, not a defect in TapeDeck

16.8 Related documentation

- CALF Gateway — Market Data — the protocol TapeDeck's bridge speaks upstream
- API Gateway — the REST history endpoints the bridge proxies
- Centralized Log Server — the bridge's optional operational-logging destination
- Configuration GUI (`config-gui`) — the sibling application TapeDeck shares its monorepo shape and deployment conventions with
- `docs-design/EduMatcher-Terminal-GUI.md` — the full design document (repository checkout only)
- `web-apps/terminal-gui/README.md` — the implementation's own record of every deviation from that design document (repository checkout only)

CHAPTER 17

Trading GUI (`pm-trading-ui`)

Learning objectives

After reading this page you will understand:

- What the Trading GUI is for, how it differs from TapeDeck, and which of the three roles it will route you to after login
- How the browser talks to the exchange — which REST calls and which of the three WebSockets each role opens, and what happens on a reconnect
- What must be running before you can log in, and how to start the app locally, from a container, or as a production build
- How API-key login and role detection work, where to find keys, and what a `READ_ONLY` key does
- Exactly which screens and actions each role gets, and where that restriction is actually enforced
- Every screen in the app: the shared shell, the all-role market screens, and the full TRADER, MARKET_MAKER and ADMIN surfaces
- Every order type the ticket supports and every field it can show, including SMP and the client order ID
- What submitting, amending, replacing and cancelling an order each do — including what queue priority you keep or lose
- How to place, watch and cancel OCO and Combo groups
- How a market maker submits, monitors, re-quotes and cancels a two-sided quote, and how to diagnose one that looks wrong
- Every ADMIN control, which are fully live today versus read-only, and why
- The five classes of error the app can show you, and which of them is *not* a rejection
- Typical end-to-end sessions for each role, the keyboard shortcuts, the power-user setting, and a troubleshooting table

17.1 What the Trading GUI is

The **Trading GUI**, system name `pm-trading-ui`, is the graphical trading terminal for the EduMatcher exchange. Its source code lives in `web-apps/trader-gui/` and its full specification is `docs-design/EduMatcher-Trading`. Unlike TapeDeck, which is a read-only market display with no login, the Trading GUI is a **write-capable** application: it is how a student places an order, a market maker manages a two-sided quote, or an instructor

runs the exchange session from a browser instead of the ALF console.

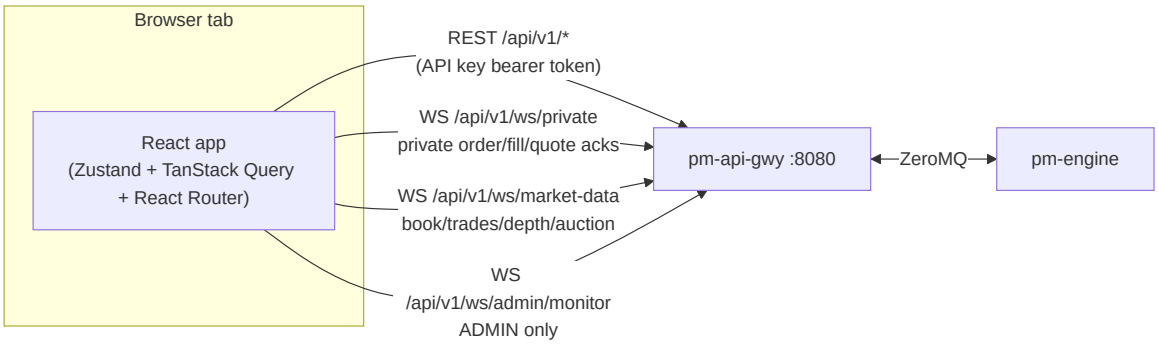
Logging in with an API key routes you to one of three role-aware personas, each with its own landing screen and navigation:

Role	Landing screen	What the role is for
TRADER	Trading Workspace	Order entry, blotter, fills, positions
MARKET_MAKER	Quote Management	Two-sided quote management, positions
ADMIN	System Dashboard	Session control, gateway management, risk, monitor log

Market Overview and Watchlist are available to every role, since watching the market is a shared need regardless of what a given key is authorised to do about it. Roles and access rights below has the full per-screen matrix.

17.2 How it talks to the exchange

The app connects to exactly one upstream service, `pm-api-gwy`, over REST and WebSocket. There is no separate bridge process — the browser talks to the API gateway directly (through a Vite dev proxy in development, or a same-origin reverse proxy / built-in proxy in production).



The three sockets are not all opened by every role:

Socket	Carries	Opened by
`/api/v1/ws/private`	Your own `order.*`, `quote.*`, `oco.*`, `combo.*` events, plus an `orders.snapshot` on every connect	**TRADER, MARKET_MAKER only**
`/api/v1/ws/market-data`	`book`, `trades`, `depth`, `auction`, `session.state`, `circuit_breaker`, `news`	Every role

Socket	Carries	Opened by
`/api/v1/ws/admin/monitor`	Cross-gateway order activity and the audit tail	**ADMIN only**

The consequence worth remembering: **ADMIN has no private order stream.** An ADMIN key sees every gateway's activity through the monitor feed, but it does not get an Active Orders blotter of its own.

Three details shape how the screens behave:

- **REST hydrates, WebSocket keeps current.** `GET /api/v1/bootstrap/{role}` fills every store at login; after that the sockets carry the changes, so most screens need no polling. The exceptions are the polled daily-stats rollup (change % / volume) and the REST-backed panels marked below.
- **`orders.snapshot` is the reconcile point.** It arrives on the first `/ws/private` frame *and on every reconnect*, so the blotter and the quote cards resync themselves after a dropped connection without you doing anything.
- **Gaps are visible, not silent.** The market-data socket tracks a sequence number per topic; the admin monitor feed marks any stretch it could not replay after a reconnect, and any events the gateway dropped under load, with a red `GAP` row in the Monitor Log.

17.3 Prerequisites

Requirement	Notes
`pm-api-gwy` (API Gateway)	Required — this is the only upstream service the Trading GUI talks to. It must be reachable on `localhost:8080` (or wherever `VITE_API_BASE`/`API_PROXY_TARGET` points).
`pm-engine`	Required indirectly — `pm-api-gwy` authenticates your gateway against it at login, so a running gateway with a stopped engine still fails login.
An API key bound to a `TRADER`, `MARKET_MAKER`, or `ADMIN` gateway identity	Defined under `api_gateways.<name>.credentials` in `engine_config.yaml`. See Logging in below.
Podman 4 or **Docker 24** with a Compose plugin	Needed for the container run path.
Node.js 22 (LTS) and npm	Needed only for local development or serving the production bundle without a container.
`pm-stats` (optional)	Powers Trade History, the daily rollup, the chart's history and the Order Detail timeline. Without it those degrade to an explanatory notice; live trading is unaffected.
`pm-audit` (optional)	Powers the ADMIN cross-gateway order drill-down only.

17.4 Running the application

There are three ways to run it, and which one you want depends on what you are doing. If you are not sure, use the first.

	What it is	Open	Use it when
The whole stack	The exchange and the five web applications started by default (`terminal-gui`, `log-gui`, `trader-gui`, `book-gui`, `dash-gui`) — a sixth, `config-gui`, is opt-in (`CONFIG_GUI=1`) since it talks to no exchange — together as containers	http://localhost:8093	You want a working system. This is almost always the right answer
This app alone, in a container	Just the Trading GUI, pointed at an exchange you started some other way	http://localhost:8093	The exchange runs elsewhere — another machine, a VM, a host install
Local dev server	Vite with hot reload, on your machine	http://localhost:8193	You are changing this application's code

17.4.1 The whole stack (recommended)

This starts the exchange *and* the Trading GUI together, on one private network, with no addresses to configure:

A released install:

```
cd ~/.edumatcher
./edumatcher.sh start
```

From a source checkout:

```
cd deployment/docker
make up-all
```

Then open **http://localhost:8093**. The exchange's REST API is reachable at `edumatcher:8080` from inside the network, which the compose file sets for you — there is nothing to point anywhere.

The other applications come up at the same time: the trader terminal on 8090, the log console on 8091, the order book viewer on 8094 and the operator dashboard on 8095.

Everyday commands, from `deployment/docker` (or with `./edumatcher.sh` in a released install):

```
make status      # is every exchange process up?
make logs        # the backend's own output
make down-all    # stop everything
```

Full detail — including how to change which configuration the exchange trades — is in Installation.

17.4.2 This app alone, in a container

Use this when the exchange is already running somewhere else. From `web-apps/trader-gui/`:

```
make up
```

Then open **http://localhost:8093**.

This path has one thing to configure, because the application is no longer on the same network as the exchange: where `pm-api-gwy` is. The compose file defaults to `host.docker.internal`, which resolves on Docker Desktop but not on Podman or Linux Docker, so set it explicitly:

```
# Podman
API_PROXY_TARGET=http://host.containers.internal:8080 make up

# Linux Docker, or a gateway on another machine
API_PROXY_TARGET=http://192.168.1.50:8080 make up
```

If port 8093 is taken on your machine, move the *host* side of it:

```
TRADER_GUI_PORT=8193 make up
```

`make logs` follows the server log, `make down` stops it, `make ps` shows the stack.

CALF_* and LOG_SRV_* in this compose file

The file sets them for parity with TapeDeck, but `apps/serve/serve.ts` does not read them: the Trading GUI has no CALF uplink and no bridge process of its own, only the REST/WebSocket proxy. Leave them alone.

17.4.3 Local dev server

```
make install    # once, and after any dependency change
make dev        # Vite with hot reload
```

Open **http://localhost:8193**, enter an API key, and click **Connect**.

`vite.config.ts` proxies every `/api/*` request — REST *and* the WebSocket upgrades under `/api/v1/ws/` (`private`, `market-data`, `admin/monitor`) — to `http://localhost:8080`. So as long as `pm-api-gwy`'s `desk` instance is reachable on port 8080 of your own machine, this needs no configuration: `VITE_API_BASE` and `VITE_WS_BASE` stay empty and the browser only ever talks to one origin, which sidesteps CORS entirely.

Starting the exchange with `make up-all` publishes 8080 on `127.0.0.1`, so the container stack and this dev server work together with nothing further to set. See The Development Loop for the full inner-loop workflow.

17.4.4 Production build and serve (without a container)

```
make build      # typecheck + vite build → apps/web/dist/
make serve      # serve apps/web/dist/ via pm-trading-ui-serve, :8093
```

`pm-trading-ui-serve` (`apps/serve/serve.ts`) is a small dependency-free Node `http` server: it serves the built SPA with `Cache-Control: immutable` on hashed asset bundles and `no-cache` on `index.html`, falls back every unknown `GET` path to `index.html` so React Router can handle client-side navigation, and optionally proxies `/api/*` to `pm-api-gwy` when `API_PROXY_TARGET` is set. With `API_PROXY_TARGET` unset, `/api/*` returns an explanatory `503` rather than silently 404-ing.

```
# Custom port, proxy API to a remote gateway
PORT=8088 API_PROXY_TARGET=http://my-exchange:8080 npm run serve
```

In production you have two options for `/api/*`: put a reverse proxy (nginx, Caddy, Traefik) in front that routes `/api/*` to `pm-api-gwy` and `/` to `pm-trading-ui-serve` (recommended, `API_PROXY_TARGET` left unset), or set `API_PROXY_TARGET` and let `pm-trading-ui-serve` forward `/api/*` itself, which needs no extra infrastructure for a single-machine setup.

17.4.5 Make targets

Target	What it does
`make install`	`npm ci` from the lockfile across all workspaces (`npm install` when there is no lockfile yet)
`make dev`	Vite dev server on `:8193` with hot reload
`make build`	Typecheck + production build into `apps/web/dist/`
`make build-debug`	Build with source maps, skipping the typecheck
`make typecheck` / `make lint`	TypeScript type-check (TypeScript *is* the linter here)
`make test`	Vitest suite
`make format`	Prettier over the whole tree
`make serve`	Serve the built SPA via `pm-trading-ui-serve` on `:8093`
`make up` / `make down` / `make restart`	Start / stop / restart this app's own container
`make cbuild`	Build the container image without starting it
`make logs` / `make ps`	Follow container logs / show stack status
`make cdist`	Build a distributable image tarball under `dist/`
`make clean`	Remove build artefacts and `dist/`
`make distclean`	Also remove `node_modules` and the lockfile, for a pristine `make install`
`make imageclean`	Remove this project's local container images
`make ghcr-login` / `cpush` / `ghcr-logout` / `ghcr-clean`	Log in to the GitHub Container Registry, push the image, log out, and log out plus remove the local images

Run `make help` for the authoritative list, `npm run serve -- --help` for the static server's own flag reference, and see `web-apps/trader-gui/README.md` for the implementation's own notes. ## Logging in

The Trading GUI has no user database of its own. It authenticates against `pm-api-gwy` using **API keys** defined under `api_gateways.<name>.credentials` in `engine_config.yaml`, which the gateway loads at startup. Entering a key and clicking **Connect** does three things in order:

1. `GET /api/v1/status` with the typed key — the gateway validates it, authenticates the bound gateway id against `pm-engine`, and returns the `gateway_role` (`TRADER`, `MARKET_MAKER`, `ADMIN`, or `READ_ONLY`).

2. `GET /api/v1/bootstrap/{role}` — hydrates every store (symbols, reference data, session phase, halts, positions, orders) *before* any socket opens.
3. Only then is the key committed to the in-memory auth store, which is what opens the WebSockets and unblocks the app shell.

That ordering matters: a key that turns out to be invalid never causes three sockets to be opened against it.

```
api_gateways:
  desk:
    credentials:
      - api_key: key-trader-demo       # ← use this in the login form
        gateway_id: TRADER01           # ← maps to participants id TRADER01
      - api_key: key-mm-demo
        gateway_id: MM01                # role: MARKET_MAKER
      - api_key: key-admin-demo
        gateway_id: OPS01                # role: ADMIN
```

`engine_config.yaml` lives in the EduMatcher session data directory — by default `~/.local/share/edumatcher/ref_` or wherever `EDUMATCHER_DATA_DIR` points. Keep one credential per persona in your config so you can switch roles by reconnecting with a different key.

READ_ONLY keys cannot use this app

A credential defined with `gateway_id: null` is a **read-only** key: the gateway reports `gateway_role: "READ_ONLY"` for it. The Trading GUI is a write-capable terminal and rejects it at login with *"`ROLE_UNSUPPORTED`: Unsupported role: READ_ONLY"*. Read-only keys are for observer tooling such as TapeDeck; use a key with a `gateway_id` here.

The key never touches disk

The API key is held in memory for the tab only (`useAuthStore`), never written to `localStorage` or `sessionStorage`. Reloading the page returns you to the login screen — expected for a classroom system running on localhost, not a bug.

17.5 Roles and access rights

Your role is a property of the **gateway identity your API key is bound to**, not of the app. It decides which sidebar entries you see and which routes you may visit.

Screen	TRADER	MARKET_MAKER	ADMIN
Market Overview			
Watchlist			
Symbol Detail overlay			
Trading Workspace		–	–
Order Entry (incl. OCO / Combo)		–	–
Active Orders blotter		–	–
Trade History		–	–

Screen	TRADER	MARKET_MAKER	ADMIN
Positions (incl. Flatten)			–
Quote Management	–		–
Quote Bootstrap & Legs	–		–
System Dashboard	–	–	
Index Administration (read-only)	–	–	
Session Control	–	–	
Risk Controls (read-only)	–	–	
Circuit Breakers	–	–	
Gateway Management + Kill Switch	–	–	
Monitor Log	–	–	

Where the restriction actually lives

The sidebar and the route guard are a **presentation-layer** convenience — they keep you out of screens your key has no business on. The real enforcement is server-side, in two places:

- **pm-api-gwy** gates every `/admin/*` endpoint on the ADMIN role (`403 ROLE_DENIED` otherwise), and gates every write on the key not being read-only (`403 READ_ONLY`).
- **pm-engine** enforces the rest. Most notably, quotes are rejected for any non-market-maker participant: *"Quotes are only allowed for MARKET_MAKER participants"*.

So a hand-crafted REST call from a TRADER key cannot become an admin action or a quote, even though the gateway itself does not distinguish TRADER from MARKET_MAKER on the order and quote endpoints.

17.6 The application shell

Every authenticated screen shares the same chrome.

17.6.1 Top bar

Left to right:

- **Wordmark** — `EduMatcher pm-trader v<version>`, the version being the release the app was built from.
- **Session-phase badge** — colour-coded: Pre-Open (slate), Opening/Closing Auction (amber), Continuous (green), Closed (red), and Unknown (grey) until the first session state has arrived after login. It animates on change. The other EduMatcher GUIs show the same badge, clock and countdown.
- **Phase clock** — a countdown to the next scheduled transition (`→ Continuous in 04:12`) when the schedule provides one, or *elapsed time in phase* when it does not, so a venue with sessions disabled or a partial schedule still gets a useful clock rather than a blank one.
- **Exchange clock** — wall time, `HH:MM:SS` (hidden on narrow windows).
- **Connection health** — `Connected` / `Reconnecting` / `Disconnected` with an icon; hover for the per-socket breakdown (`events: … · market-data: … · monitor: …`).
- **"Updated HH:MM:SS"** — the timestamp of the last market-data frame, so a frozen board is obvious.
- **Command palette** (magnifier, `Ctrl+K`).

- **Notification bell** with an unread count badge (99+ above 99).
- **Settings** (gear) — cancellation confirmations, font size and the symbol picker; see Settings.
- **Theme switch** (sun / moon) — toggles between the dark and light palettes. The icon shows the theme you would switch *to*; the choice is remembered in this browser.
- **Help** (Ctrl+/).
- **Gateway id and role**, then **Logout**.

17.6.2 Left sidebar

Persistent and not collapsible. Market Overview and Watchlist are listed for every role; below a divider, the role-specific screens from the matrix above.

17.6.3 Connection banner

A thin strip appears under the top bar whenever the live connection degrades: amber *"Reconnecting to the exchange… live data is paused."* or red *"Disconnected from the exchange — is pm-api-gwy / the engine running? Live prices and events are stale until the connection returns."* It makes a stalled gateway visible everywhere rather than leaving individual screens silently stale.

17.6.4 App-level overlays

Six overlays can open over any screen. Only one drives the main content area at a time; Escape closes whichever is open.

Overlay	Opened by	Purpose
Symbol Detail	Clicking a Market Overview / Watchlist row, or a command-palette symbol	Chart, depth, tape, stats, auction and company profile for one symbol
Event Center	Bell icon, Ctrl+.	Session history of acks, fills, rejects, cancels, CB and session events
Order Detail drawer	Double-click a blotter row, an Event Center entry, or a Trade History order id	One order's full chronological lifecycle
Help drawer	? button, Ctrl+/	Topic-based in-app help
Shortcuts dialog	? (outside a text field)	The keyboard reference card
Command palette	Ctrl+K	Fuzzy search over symbols and role-aware actions

17.6.5 Event Center

A right-edge sheet listing this session's notifications, newest first, capped at 500 entries. Each row carries a coloured kind badge — ACK, FILL, REJECT, CANCEL, CB, SESSION, SYSTEM — a timestamp, a title and a detail line. Opening the sheet marks everything read and clears the bell badge. Filter chips at the top narrow to one kind (only kinds actually present are offered), **Clear** empties the list, and any entry that carries an order id is clickable straight through to the Order Detail drawer.

The Event Center is deliberately not a duplicate of the toast stream: the order ticket surfaces its own synchronous accept/reject verdict, and the event bridge covers the *later* outcomes — fills, cancels, expiries, and OCO / combo group events.

17.6.6 Command palette

`Ctrl+K` opens a keyboard-first fuzzy search over two groups:

- **Symbols** (up to 8 matches) — showing the live last price, with a star that toggles watchlist membership without leaving the palette. Selecting a symbol sets it active and opens Symbol Detail.
- **Actions** — every navigation target for your role, plus Flatten All (TRADER/MM), the Event Center, Help and the shortcut card. Rows show their keyboard shortcut where one exists.

`↑`/`↓` move, `Enter` runs, `Escape` closes. This is the backbone of mouse-free navigation.

17.6.7 Help drawer

`Ctrl+/` (or the `?` icon) opens a right-edge sheet with a topic list: Getting Started, Trading Workspace, Order Types, Amend vs Cancel-Replace, Time in Force, Auctions & Indicative Price, Risk Controls, Market-Maker Quoting, Admin Reference, and Keyboard Shortcuts. The shortcuts topic renders the same table as the standalone `?` dialog, so the two can never drift apart.

17.6.8 Error boundary

The routed screen is wrapped in an error boundary keyed on the URL. If one screen throws, it degrades to an inline message while the top bar, sidebar and overlays stay usable — and navigating away clears it.

17.7 Screens available to every role

17.7.1 Market Overview

The reference board, available regardless of role. Columns:

Column	Source
	Client-only watchlist toggle
Symbol	Reference data
Bid / Ask / Last	Live `book` and `trades` channels; flash on change
Chg %	Polled daily rollup (`/history/daily`) versus the day's open
Volume	Polled daily rollup, topped up live between polls
Status	`HALT` / `HALT <level>` badge, or an `AUCTION` badge with the indicative uncross price during a call phase

Live prices come from an always-on broad book/trades subscription; the derived columns come from the polled rollup and **degrade gracefully** — if the stats database is unavailable you get *"Daily rollup unavailable — Chg %/Volume hidden"* in the header and blank cells, not a broken table.

The table is virtualised, so a few hundred symbols scroll smoothly. Every column except and Status is sortable; symbols with no known open price sort *last* in both directions rather than heading an ascending sort. The filter box narrows by symbol substring, and **Refresh** re-fetches `/symbols` and the rollup on demand.

Clicking a row opens the Symbol Detail overlay and sets that symbol as the app-wide **active symbol** — which the Trading Workspace, the order ticket and the focused market-data subscription then follow. The star toggles the watchlist without selecting the row.

Two badges are worth reading carefully:

- **`HALT`** with no level is an *administrative* halt (an ADMIN triggered it with no ladder level); `HALT L2` names the circuit-breaker level that fired.
- **`AUCTION`** shows the engine's equilibrium price, with a trailing `~` while the value is still indicative. `no cross` is a real reading — the book would not cross at any price — not a missing value.

17.7.2 Watchlist

`Ctrl+L`. A thin wrapper around the shared watchlist panel: a compact board of your starred symbols showing Last, Chg %, Bid and Ask. Row click behaves like Market Overview; the star removes the symbol.

The watchlist is more than cosmetic: **the watchlist set drives the market-data focus subscription**. The active symbol plus your watchlist are the symbols the heavy `depth` and `auction` channels are requested for, so starring a symbol is how you get full depth on it without making it active. The active symbol leads the set so it survives if the set hits `VITE_MAX_FOCUS_SYMBOLS`.

The watchlist is client-only and lives in memory for the tab — it is not persisted between sessions.

17.7.3 News

The headlines `pm-market-sim` publishes (see The market model), newest first: time, status, what the headline is about (symbols, a sector, or `MARKET`) and the headline itself, green for good news and red for bad. The screen loads the latest headlines when the market-data socket connects (and again on every reconnect); new ones arrive live.

- **`RUMOUR 60%`** marks an unconfirmed headline; the percentage is its credibility, how believable it is. Nothing has happened to the company yet.
- When a rumour resolves, a second headline arrives marked **`CONFIRMED`** or **`RETRACTED`**. A retracted rumour and its retraction are both struck through.
- The filter box takes a symbol or a sector name. A symbol shows its own news, its sector's news and market-wide news; a sector shows its own news and its symbols' news.

The screen shows the latest 50 matching headlines; the gateway keeps 200.

17.7.4 Symbol Detail overlay

A 640 px right-edge panel with a live header (symbol, last, change %, volume) and six tabs. The Auction tab shows an amber dot while a call phase is running.

Tab	Contents
Chart	Candlestick or line chart. Timeframes `1m`, `5m`, `1h`, `1D`, `All`. Intraday frames are built from trade prints and **append live** as trades arrive; `1D`/`All` render the daily rollup and do not live-append.
Depth	The full depth ladder — see below.
Trades	The last 50 prints, seeded from `/history/trades` and topped with the live tape, de-duplicated by trade id. Price is green when a buyer was the aggressor, red when a seller, grey `A` for an auction print.
Stats	Today's OHLCV grid: Open, High, Low, Close/Last, Volume, Trade Count, VWAP, Largest Trade, Last Buy, Last Sell.
Auction	The engine's indicative uncross (or the most recent completed result): equilibrium price, matched quantity, imbalance side and quantity — plus a **cumulative supply/demand curve** derived from the resting book, with the equilibrium price marked.
Company	The company behind the symbol, from the venue's company database: its name (highlighted), sector, industry, headquarters and purpose, then **Valuation** (market cap, P/E, P/S, dividend yield, debt/equity, credit rating), **Financials** for the last fiscal year (revenue and its growth, net income, net and operating margin, free cash flow, in millions of USD), **Position** (key market, market position, competitors, key risk), **Management** (CEO and tenure, years in the industry, employees) and **History** (founded, IPO, origin, founder story). Without a company database the tab says so.

The depth ladder

Bids on the left, asks on the right, each level showing order count, quantity and price, with a bar behind the quantity proportional to that level's share of the deepest level in view. A `5` / `10` / `20` selector sets how many levels to show.

Click-to-trade: clicking a *bid* level pre-fills a **SELL** at that price; clicking an *ask* level pre-fills a **BUY**. In the Trading Workspace this writes straight into the order ticket (switching it to `LIMIT`, filling the price, and ring-highlighting the suggested side button).


EduMatcher pm-trader v0.50.7
Market Overview
Watchlist
News
Trading Workspace
Order Entry
Active Orders
Trade History
Positions
Workspace
Symbol
AAPL
1m
5m
1h
1D
All
17:32
17:50
Order Ticket
AAPL
Market
Limit
Stop

17.8 The TRADER role

17.8.1 Trading Workspace

The shell: the top bar (session badge and clocks, connection health, notifications, settings, theme, help, gateway id and role, logout) and the TRADER sidebar. The four panels follow the active symbol: the chart in one-minute candles, the depth ladder (click a price to pre-fill the ticket), the compact ticket with the symbol locked, and the working AAPL orders, where the iceberg is `PARTIAL` *with 157 of 400 left.*

The default TRADER landing screen and cockpit. Four panels, all bound to one active symbol chosen with the symbol picker in the header — a dropdown, or a type-to-filter search box on an exchange with many symbols (see Symbol picker):

- **Chart** (top-left, spanning two columns) — the same chart component as Symbol Detail.
- **Depth ladder** (right, full height) — click-to-trade into the ticket.
- **Order ticket** (below the chart) — in **compact** mode with the symbol **locked** to the active symbol and the client-order-id field hidden.
- **Compact blotter** (bottom strip) — this symbol's *working* orders only (terminal rows are filtered out), with an inline cancel per row. It reads the same live order store as the full blotter, so it needs no polling.

Changing the symbol re-binds every panel at once. If no symbol is active yet the Workspace adopts the first known one so all four panels have something to show; with no symbols at all it says so and points at `pm-api-gwy`.

17.8.2 Order Entry

The standalone full ticket, with the symbol picker **unlocked** — choosing a symbol here also sets the app-wide active symbol. Below it, an **Advanced** disclosure holds the **OCO** and **Combo** sub-forms, collapsed by default so a first classroom session stays simple.

The eight single-leg order type tabs are always shown flat above the ticket fields — every single-leg type is one click away. Only OCO and Combo sit behind the Advanced disclosure.

Ticket fields

Every field carries a small button with an inline explanation, so the ticket is self-documenting in a classroom.

Field	Shown for	Notes
Symbol	Always	A combobox backed by the configured symbol list. Locked and read-only in the Workspace.
Quantity	Always	Positive whole number. Defaults to 100.
Price	LIMIT, STOP_LIMIT, FOK, ICEBERG, IOC	The placeholder shows a live **reference price** (`Ref: 150.25`) derived from last trade → mid → previous close, so you are not typing blind.

Field	Shown for	Notes
Stop price	STOP, STOP_LIMIT	The trigger price.
Visible qty	ICEBERG	The slice shown on the book. Must be strictly less than the total quantity.
Trail offset	TRAILING_STOP	Distance the stop trails behind the price.
TIF	All types except IOC	DAY / GTC / ATO / ATC — gated by session phase, see below.
SMP	Always	Self-match prevention action.
Client Order ID	Order Entry only (hidden in the compact Workspace ticket)	Optional idempotency key, up to 64 characters.

SMP (self-match prevention) deserves a note, because the default is not `NONE`. The selector offers *Gateway default*, `NONE`, `CANCEL_AGGRESSOR`, `CANCEL_RESTING` and `CANCEL_BOTH`. Leaving it on **Gateway default** omits the field entirely, which lets `pm-api-gwy` apply its own configured policy — that is deliberately *different* from explicitly choosing `NONE`, which disables SMP for this one order. Leave it alone unless you are demonstrating a specific policy.

TIF is phase-gated. The values valid in the current phase are selectable; the rest are shown greyed as `(n/a this phase)`:

Phase	Selectable TIF
`PRE_OPEN`	DAY, GTC
`OPENING_AUCTION`	DAY, GTC, **ATO**
`CONTINUOUS`	DAY, GTC
`CLOSING_AUCTION`	DAY, GTC, **ATC**
`CLOSED`	*(none — no orders accepted)*

If the phase changes while you have an illegal TIF selected, the ticket auto-corrects to the first allowed value rather than letting you submit something the engine will reject.

Order types

Type	Price	Stop price	Visible qty	Trail offset	TIF	Rests on the book?
MARKET	–	–	–	–		Never — sweeps the opposite side, unfilled remainder is discarded
LIMIT		–	–	–		Yes — fills at price-or-better, remainder rests
STOP	–		–	–		Dormant until `stop_price` triggers, then becomes a MARKET order
STOP_LIMIT			–	–		Dormant until `stop_price` triggers, then becomes a LIMIT order
FOK (fill-or-kill)		–	–	–		Never — all-or-nothing, rejected immediately if it can't fill in full

Type	Price	Stop price	Visible qty	Trail offset	TIF	Rests on the book?
ICEBERG		–		–		Yes — a LIMIT order that only shows `visible_qty` at a time; each replenished slice gets a new queue timestamp
IOC (immediate-or-cancel)		–	–	–	*(hidden — always immediate)*	Never — fills what it can now, cancels the remainder
TRAILING_STOP		–	–			Dormant; `stop_price` ratchets by `trail_offset` as the market moves, then becomes a MARKET order on trigger

Notes worth knowing before you place an order:

- A **STOP** triggers when the last trade price crosses `stop_price` (`>=` for a BUY stop, `<=` for a SELL stop). **STOP_LIMIT** triggers the same way but converts to a LIMIT at your given price instead of

a MARKET order.
- A **TRAILING_STOP** needs a reference to trail from. If the symbol has never traded and you supply no stop price, the engine rejects it with *"Trailing stop requires STOP= or a prior trade price"*.
- **FOK**'s pre-check for "can this fill in full right now" excludes hidden iceberg reserve and any same-gateway liquidity that self-match prevention would filter out — so a FOK can reject with *"Insufficient liquidity"* even when the visible book total looks sufficient.
- **IOC requires a price** — it isn't a bare sweep like MARKET, it's a price-limited immediate execution with no resting remainder.
- **MARKET, FOK and IOC are the "cannot rest" types**, and that single fact explains most of their special-casing: they are rejected outright during either auction phase and on a halted symbol, because in both cases the engine is not matching continuously and there is nowhere for them to go. Every other type is accepted and rests without matching until matching resumes. During an auction the ticket disables those three tabs and shows a banner; if one of them is selected when the auction starts, the ticket falls back to `LIMIT` rather than dead-ending.
- See Order Types for the full engine-level reference.

OCO orders

OCO (one-cancels-other) links two legs — each its own order, same symbol, opposite or same side as you configure, one LIMIT or STOP each — under a shared `oco_id`. The form carries a shared symbol, quantity and TIF (DAY or GTC), plus a per-leg side and type; an OCO id is generated for you (`oco-<base36>`) but stays editable, and a fresh one is generated after each successful submit.

When either leg reaches a **terminal state** (filled, cancelled, or rejected), the engine automatically cancels the other, with reason *"OCO sibling `{id}` reached `{STATUS}`"*, and the Event Center logs an `oco.cancelled` entry. A **partial** fill on one leg does *not* trigger the cascade — only a terminal state does.

Combo orders

Combo (multi-leg, all-or-none) links two to ten legs, potentially across different symbols, each LIMIT or MARKET, under a shared `combo_id`. The leg builder lets you add and remove legs (the remove button is disabled at two legs), and a MARKET leg's price field is disabled since it must not carry one. TIF is shared across all legs; SMP is fixed at `NONE` for combos submitted from this GUI.

Combos from this GUI are always all-or-none (`AON`): the combo is only considered matched once every leg has fully filled. If any leg cancels or expires, the engine cascade-cancels the remaining unfilled quantity on the other legs — legs that already filled are **not** reversed. A leg rejected at submission takes the whole combo down with a reason naming the offending leg (*"Leg 2 (MSFT) …"*).

`Ctrl+Enter` submits whichever of the two forms has focus.

Groups in the blotter

Both OCO and Combo legs show up as ordinary rows in the Active Orders blotter — each has its own order id, its own status, and its own Amend / Replace / Cancel buttons — with a **Group** badge linking them together, and a separate **Groups** panel above the main blotter. Each group row shows its kind (amber `OCO`, blue `COMBO`), id, member symbols, an aggregate status (`1 live / 1 cancelled`, or just `filled` when uniform) and a live/total count, with a one-click **Cancel group** that is disabled once no member is live.

17.8.3 Submitting an order

Pick a type tab, fill in the fields it shows, and press the **Buy** or **Sell** button (or the `B` / `S` shortcut, ignored while typing in a form field). Before anything is sent, the ticket checks:

- The market isn't `CLOSED` (blocked client-side with *"Market is closed — no orders accepted"*).
- MARKET, FOK, and IOC aren't being submitted during an auction phase (blocked client-side with *"`{TYPE}` orders are not accepted during an auction"*).
- Every required field for the selected type is present and valid — see Errors and how to read them for the exact messages.

If validation passes, the ticket submits with `wait=ack`, which asks the gateway to hold the HTTP response open until the first acknowledgement comes back, so you get a synchronous accepted/rejected answer rather than having to watch the blotter to find out:

- **Accepted** — a success toast (*"`{side}` `{qty}` `{symbol}` accepted"*) and an `ACK` entry in the Event Center; the order appears in Active Orders with status `NEW`.
- **Rejected** — an error toast (*"REJECTED: `{reason}`"*, using whatever reason string the engine returned) and a `REJECT` entry in the Event Center; no row appears in the blotter, since a rejected order was never accepted onto the book.
- **Timed out waiting for the ack** — this is the one case that is *not* a rejection: the order reached the engine, but the synchronous ack didn't return in time. The ticket shows *"`{side}` `{symbol}` submitted — awaiting confirmation (check blotter)"*; check Active Orders a moment later to see whether it landed.

On a successful submit, the ticket keeps your symbol, quantity, and price so you can immediately submit the other side of a two-sided view — only the client order ID field clears.

17.8.4 Active Orders

*TRADER01's seven working orders: limits, an iceberg, a stop and a stop-limit, DAY and GTC. The iceberg is `PARTIAL`; the stop shows – for its price, since a stop carries none. One row is ticked, so the bar below offers **Cancel all selected**; each row has Amend, Cancel-Replace and Cancel.*

The full live blotter, driven by the order store — seeded from the `orders.snapshot` frame and folded forward from live `order.*` events, so it never polls. The header shows the order count and a "reconciled HH:MM:SS" stamp; **Refresh** performs an explicit `GET /orders` reconcile against the gateway, merging rows without resurrecting anything already terminal locally.

Columns

Column	Notes
	Selection checkbox; disabled for terminal rows
Symbol, Side, Type, TIF	Side is colour-coded green/red
Qty	The order's total quantity
Remaining	Flashes on every change, so a fill or amend is visible at a glance in a busy blotter
Price	– for types that carry no price
Group	The `oco_group_id` or `combo_parent_id` badge, or –

Column	Notes
Status	Coloured status pill
Updated	`HH:MM:SS.mmm`
(actions)	Amend (pencil), Cancel-Replace (repeat), Cancel (×) — all greyed for terminal rows

Every column except the checkbox, Group and the action cell is sortable.

Selecting rows and moving around

With the mouse: click selects a row; **Shift-click** selects an inclusive range; **Ctrl/ -click** toggles one row without clearing the rest; the header checkbox selects every *cancellable* row. Terminal orders (filled, cancelled, rejected, expired) can never be selected.

The blotter is also fully keyboard-driveable — click a row once (or Tab into the table) and then:

Key	Effect
`↑` / `↓`	Move between rows. This moves the **focus outline only** and deliberately leaves the selection alone, so you can arrow through a busy blotter without disturbing a multi-row selection you have already built up. It stops at the first and last row rather than wrapping.
`Ctrl+A` / `⌘+A`	Select every cancellable row — the keyboard equivalent of the header checkbox. Terminal rows are skipped.
`Enter`	Open the Order Detail drawer for the focused row (same as double-click)
`Delete` / `Backspace`	Cancel the current selection, or — when nothing is selected — the focused row

The combination worth learning is `↑`/`↓` to the order you want, then `Delete`: because `Delete` falls back to the focused row when no row is selected, that is the fastest way to pull a single order without touching the mouse.

Terminal orders stay visible for a while

The blotter does not immediately drop a filled or cancelled order — the row remains with a terminal pill and greyed action buttons so you can see what happened. Up to 200 terminal orders are retained (oldest dropped first); a fresh `orders.snapshot` on reconnect removes the ones the gateway has since evicted. Working orders are never dropped.

17.8.5 Amending an order

Click the pencil icon on a working order to open **Amend**. The dialog shows symbol, side, type, TIF, original quantity and filled quantity read-only, and lets you edit **price** (when the type has one) and **quantity**. It

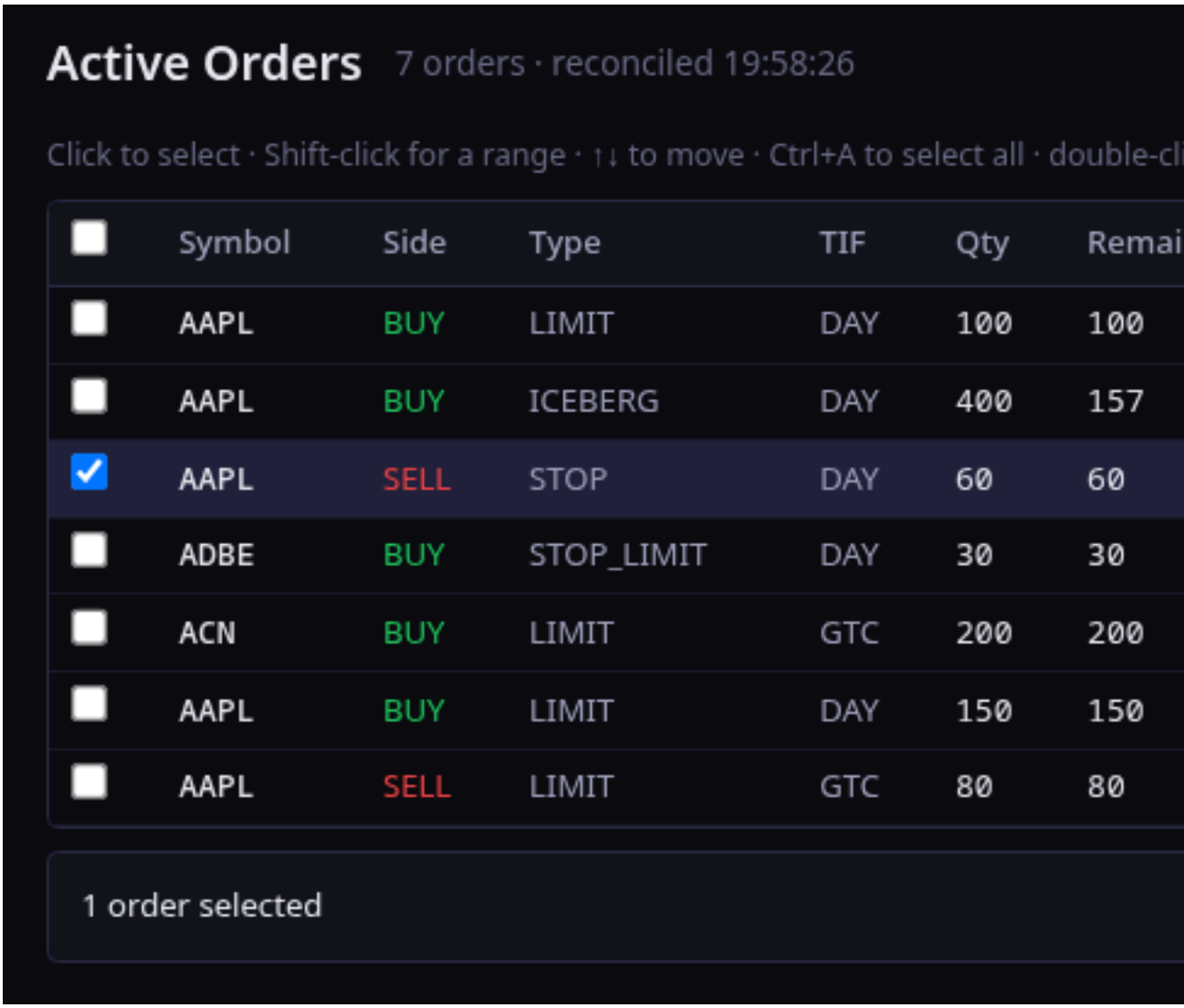


Figure 17.2: Active Orders with one order selected.

does not block a price change or a quantity increase, but it warns you about the priority consequence, and it applies the engine's own rules locally so an amend the engine would refuse never leaves the browser: the quantity must be a positive whole number that exceeds what has already filled, the price must be positive, and at least one of the two must actually change (*"No changes to submit"*). Only the fields you changed are sent.

Which change you make matters for your place in the queue:

Amendment	Queue priority
Quantity decrease only, price unchanged	**Preserved**
Price change, either direction	**Lost**
Quantity increase, price unchanged	**Lost**
Both price and quantity changed	**Lost**

In other words: a same-price size reduction is the only amendment that keeps your place in line. Anything else re-queues the order behind resting orders at the same price — the engine assigns a fresh arrival sequence — even though the order id itself doesn't change. If a price amendment makes the order marketable (it now crosses the book), the engine fills it immediately as part of processing the amend: you'll see an `order.amended` event followed by one or more fills on the same order id.

Only **LIMIT and ICEBERG** orders can be amended. MARKET, FOK and IOC never rest long enough, and STOP / STOP_LIMIT / TRAILING_STOP have ambiguous trigger semantics once amended, so the engine rejects the attempt with *"Cannot amend `{TYPE}` orders"*. The dialog itself doesn't stop you trying.

You cannot amend down to exactly what has filled

The new quantity must **exceed** the filled quantity, not merely reach it. Amending a 100-lot with 40 filled down to 40 is refused — the dialog stops it with *"Quantity must exceed the 40 already filled — cancel the order instead"*, which is also precisely what the engine would have answered. If your intent is to stop trading the remainder, cancel the order; the 40 already filled are yours either way.
An amend does not wait for the engine: the dialog closes once the gateway has taken it, and the outcome (amended, or refused with its reason) arrives in the blotter and the notifications. An error shown in the dialog means the amend was not sent.

17.8.6 Cancel-Replace

Click the replace icon to open **Replace**, which lets you change price, stop price, and quantity for a resting order, same as Amend — but the mechanism is different: Replace atomically cancels the resting order and submits a brand-new one with your changes, rather than modifying the existing order in place. Symbol, side, type and TIF are inherited and shown read-only.

Priority is always lost, regardless of what you changed, since the replacement is a new order with a fresh timestamp. Use Replace when you want a genuinely new order (for example, increasing size well beyond the original), and Amend when a same-price size reduction is enough — Amend is the only path that preserves your queue position.

On success you get *"Replaced `{old}` → `{new}`"*, naming both order ids.

A Replace can fail in two ways, and the dialog stays open and says which:

- *"Replace refused: {reason} — nothing was sent."* The engine refused to cancel the original, typically because it filled first. No replacement was submitted.
- *"The cancel was not confirmed and no replacement was sent — check the blotter: the original may still be working."* The cancel's outcome did not come back in time. The gateway never sends a replacement for an unconfirmed cancel, so you cannot end up with two orders, but the original may still be live.

17.8.7 Cancelling an order

Click the × icon on a row, or select it and press Delete/Backspace, to cancel a single order. With **Confirm cancellations** on (the default), a confirmation dialog appears first. With it off, the cancel fires immediately and you get an undo toast instead — **Undo** resubmits an equivalent order at the *remaining* quantity, but as a brand-new order id with no priority carried over (the toast says so explicitly: *"Undo re-submits an equivalent order (priority not preserved)"*). If nothing remained, the undo is a no-op and says so rather than sending a zero-size order.

The same behaviour applies to the compact blotter in the Trading Workspace — both use the same shared cancel path, so power-user mode behaves identically in either place.

Bulk cancel: select multiple rows and use the **Cancel all selected** button that appears, or press Delete with a selection active. This confirms once for the whole batch (*"…This cannot be undone."*), then cancels each selected order individually — it is not a single mass-cancel call, so a very large selection sends one request per order.

Group cancel: the Groups panel above the blotter has a one-click cancel per OCO/Combo group, which cancels the group as a unit (DELETE /oco/{id} or DELETE /combos/{id}) rather than looping over its member orders. The confirmation spells out the consequence, including that a combo's already-filled legs are not reversed. The outcome arrives as a notification, *"OCO {id} cancelled"* or *"Cancel of OCO {id} refused"* (likewise for combos), next to the members' own cancels in the blotter.

17.8.8 Seeing whether an order is resting or executed

Every order has one of these statuses, shown as a coloured pill:

Status	Meaning	Terminal?
NEW	Accepted, resting, unfilled	No
PARTIAL	Resting, partially filled	No
FILLED	Fully filled	Yes
CANCELLED	Cancelled, whatever remained is gone	Yes
REJECTED	Never accepted onto the book	Yes
EXPIRED	TIF elapsed (e.g. a DAY order at session close)	Yes
PENDING	Submitted, ack not yet received (transient, client-side only)	No

NEW and PARTIAL are the two "resting" states — the order is still working and its Amend / Replace / Cancel buttons are active. The four terminal statuses grey out those buttons and can't be selected for bulk cancel.

17.8.9 Order Detail drawer

For the full chronological picture of one order, double-click its row (or press `Enter` with it focused) to open the **Order Detail drawer**. Its header repeats the symbol, short order id, status pill, side, type, TIF, quantity and — when present — the client order id and any OCO / combo group id.

The timeline shows every `ACK`, `REJECT`, `FILL`, `AMEND`, `CANCEL` and `EXPIRE` event for that order id in order, each with a `HH:MM:SS.mmm` timestamp and a detail line (`120 @ 150.25 · 80 left`, `price 151.00 · qty 200 · priority reset`, and so on). It is **seeded from the durable history** in `pm-stats` and **appended live** for the rest of your session, with live entries flagged; the two together stay accurate even when the history endpoint lags the live stream. Without `pm-stats` you get *"History unavailable — the stats database is not running. Live events below still update."*

The same drawer is opened from Trade History order ids and from Event Center `FILL`/`CANCEL` entries, so there is only ever one lifecycle view to learn.

17.8.10 Trade History

Durable fills from `GET /history/fills`, with live `order.fill` events prepended as they arrive this session (flagged `live`, and only shown when the date filter is today or unset).

Filters: **symbol** (server-side), **date** (server-side, defaults to today) and **side** (client-side). Columns are Time, Symbol, Side, Fill Qty, Fill Price, Remaining, Trade ID and Order ID. A fill that swept several price levels shows the first trade id with a `+N` badge for the rest. Clicking the Order ID opens the Order Detail drawer.

Without `pm-stats` the page shows *"Could not load fills — is the stats DB available?"*; live fills for the current session still appear.

17.8.11 Positions and Flatten

The shared Position Summary Panel, also used by MARKET_MAKER. Net position per symbol from `GET /positions`, refreshed whenever an `order.fill` arrives. Columns: Symbol, Position (signed, colour-coded), Last Price (live where available, else the cached close), Action.

Flatten submits a MARKET closing order for the whole position — a SELL for a long, a BUY for a short, at `abs(net_qty)`. **Flatten All** does the same for every non-zero position.

Because MARKET orders are only accepted during continuous trading, both actions are **disabled outside** **`CONTINUOUS`**, with an amber banner naming the current phase.

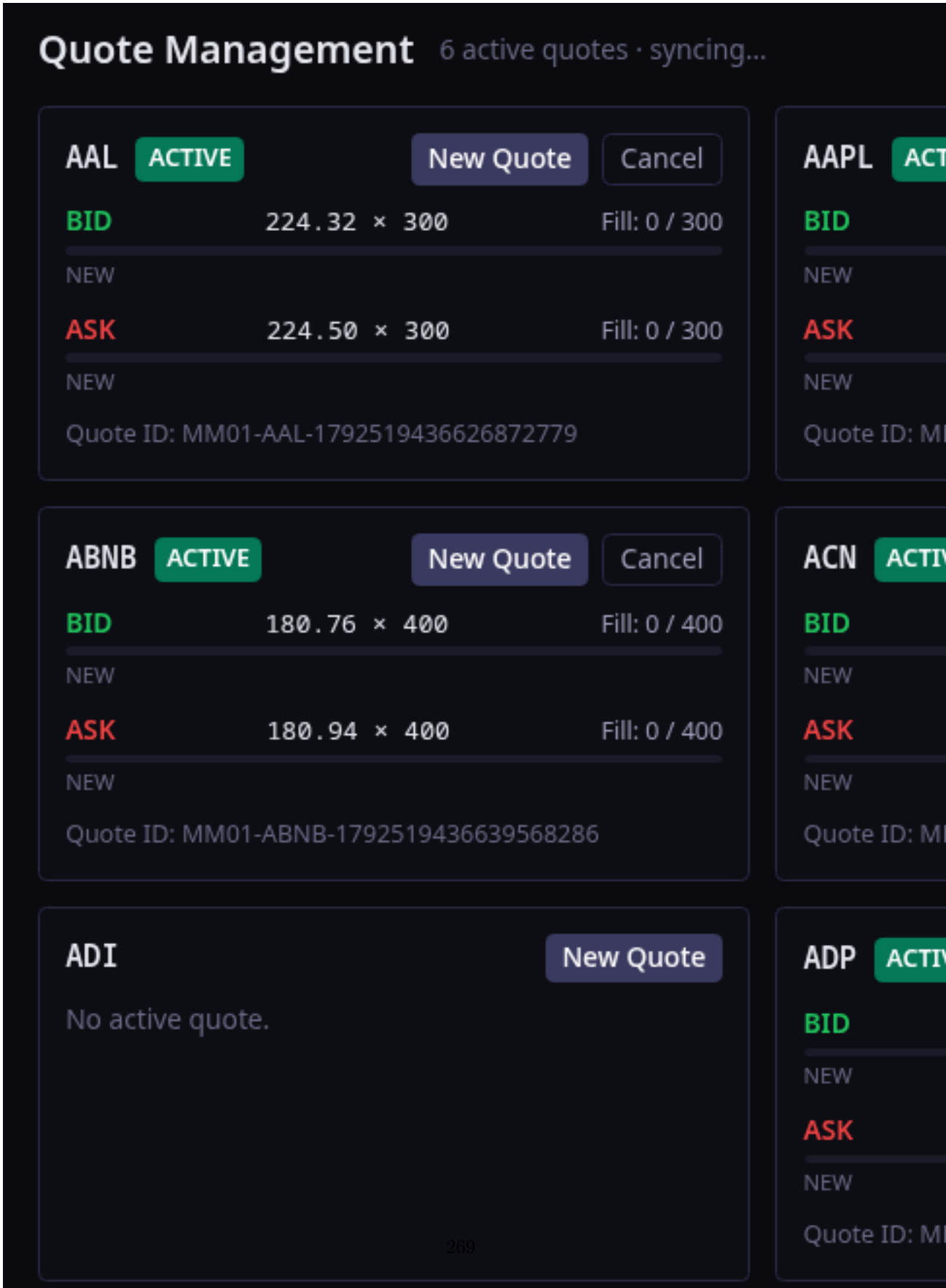
Confirmation behaviour differs between the two on purpose:

- **Flatten (per row)** honours the power-user setting: confirm dialog by default, or fire-immediately with an **Undo** toast when confirmations are off. That undo is best-effort — it cancels the just-submitted MARKET order, which does nothing if it already filled.
- **Flatten All** *always* confirms, regardless of the setting, and names how many positions it will close.

17.9 The MARKET_MAKER role

17.9.1 Quote Management

One card per symbol, in alphabetical order. Six carry an `ACTIVE` *two-sided quote: price × quantity per leg, the fill counter and progress bar, the leg status and the quote id. ADBE is halted, and the quotes on ADI and*


Quote Management
6 active quotes · syncing...
AAL
ACTIVE
New Quote
Cancel
BID
224.32 × 300
Fill: 0 / 300
NEW
ASK
224.50 × 300
Fill: 0 / 300
NEW
Quote ID: MM01-AAL-1792519436626872779
AAPL
ACT
BID
NEW
ASK
NEW
Quote ID: MI
ABNB
ACTIVE
New Quote
Cancel
BID
180.76 × 400
Fill: 0 / 400
NEW
ASK
180.94 × 400
Fill: 0 / 400
NEW
Quote ID: MM01-ABNB-1792519436639568286
ACN
ACTIV
BID
NEW
ASK
NEW
Quote ID: MI
ADI
New Quote
No active quote.
ADP
ACTIV
BID
NEW
ASK
NEW
Quote ID: MI

ADSK were hit and taken off, so those cards read No active quote.* The header counts the active quotes and shows `syncing…` while the bootstrap refetches.*

The MARKET_MAKER landing screen: **one card per configured symbol**, whether or not you currently quote it. A symbol carries at most one active quote per gateway, so each card shows at most one.

A card with an active quote shows:

- A **state badge** — `ACTIVE` (green), `INACTIVE_BID_FILLED` / `INACTIVE_ASK_FILLED` (amber), `PENDING` (slate), or `CANCELLED` / `MISSING`.
- A **BID row** and an **ASK row**, each with `price × qty`, a `Fill: filled / qty` counter, a **fill-progress bar**, and the per-leg status underneath.
- The **Quote ID**.
- **New Quote** and **Cancel** buttons.

A card with no active quote shows *"No active quote."* and just the New Quote button.

The header reports how many active quotes you have and flags `syncing…` while the bootstrap query refetches. Pressing **F2** anywhere on this screen opens the New Quote form for the current active symbol and focuses it.

17.9.2 Submitting a quote

New Quote opens the form inline on the card. Opening it also sets that card's symbol as the app-wide active symbol, so your chart and depth follow what you are quoting.

Field	Notes
Quote ID	Auto-generated as `mm-<symbol>-<base36>`, editable, auto-focused and pre-selected on open so you can type over it
Bid price / Bid qty	Defaults to 500 qty
Ask price / Ask qty	Defaults to 500 qty
TIF	DAY or GTC

A live **spread indicator** sits beside the TIF selector and updates as you type: `Spread: 0.05 (5 ticks)`, using the symbol's own tick size. It reads `Spread: –` until both prices are valid and the ask exceeds the bid.

The form refuses to submit a crossed or locked quote: *"Ask price must be strictly greater than bid price"*. On success you get *"Quote `{id}` submitted"* and an `ACK` entry in the Event Center.

Opening New Quote on a card that already has a quote **pre-seeds the form with the current prices and sizes** (with a fresh id), so adjusting a live quote is a two-field edit rather than a re-entry.

17.9.3 Watching and reacting to fills

Quote lifecycle events arrive on the `/ws/private` socket and drive three things:

- **`quote.ack`** — a rejected quote raises an error toast (*"Quote `{symbol}` rejected: `{reason}`"*) and a `REJECT` entry; an accepted one is recorded quietly as an `ACK`.
- **`quote.status`** — when a leg fills, the quote goes inactive on that side and you get a success toast, *"`{symbol}` `BID`/`ASK` filled — quote inactive"*, carrying a **Re-quote** action.

- Either event, plus every `orders.snapshot` (so, on connect and every reconnect), invalidates the bootstrap and legs caches so the cards resync to engine truth.

Re-quote re-opens the New Quote form pre-filled with the previous quote's prices and sizes, so you can decide whether to hold the same market or adjust before resubmitting.

17.9.4 Cancelling a quote

Cancel on a card removes the quote via `DELETE /quotes/{symbol}`, which takes **both resting legs** at once.

Quote cancels always confirm

Unlike a single-order cancel, the quote cancel **always** shows a confirmation dialog — the power-user *Confirm cancellations* setting does not apply to it, and there is no undo toast. A quote cancel removes two resting orders at once, so it is treated as a high-impact action.
The outcome arrives as a notification: *"Quote `{symbol}` cancelled"*, or *"Cancel of quote `{symbol}` refused"* with the engine's reason (for example when there was no live quote).

17.9.5 Quote Bootstrap & Legs

A diagnostic screen with two complementary read sources and a **Resync** button that refetches both, plus a "Reconciled at" stamp.

Active Quotes (`GET /quotes`) is the **reliable per-side source**: symbol, quote id, state, bid and ask as `price × qty (remaining)`, and per-leg status.

Legs (`GET /quotes/legs`) is the granular view: symbol, quote id, order id, side, price, qty, remaining, filled, leg status and quote status.

This endpoint is deliberately dual-shaped, and understanding that is the point of the screen. On an engine round-trip it returns full per-leg records. But immediately after a quote event lands, the gateway serves it from its live cache, which carries only **quote-level** ack/status information — so those rows come back with the per-leg columns blank. When any such row is present the page prints an explanatory note pointing you back at the bootstrap table for authoritative per-side price and quantity. It is a degraded shape, not a bug and not lost data.

17.9.6 Positions

The same Position Summary Panel as TRADER, with the same Flatten / Flatten All behaviour — see Positions and Flatten above.

17.10 The ADMIN role

17.10.1 System Dashboard

The ADMIN landing screen — a read-only overview fed by the `/api/v1/ws/admin/monitor` WebSocket (cross-gateway orders and events) plus the shared live market stores.

- **Four KPI cards**: current session phase, active orders across *every* gateway, connected gateway count, and active circuit-breaker halts (turning amber when non-zero).
- **Per-symbol summary**: Symbol, Bid, Ask, Last, Volume, Orders (that symbol's live order count across all gateways) and CB status.

- **Recent Events**: the last 15 cross-gateway events with kind badges. Any row carrying an order id is clickable through to the audit drill-down.

If the monitor feed itself is degraded, an amber *"Monitor feed reconnecting/disconnected"* note appears next to the page title.

17.10.2 Index Administration

Read-only. Configured indexes (id, description, base value, constituent count — hover for the full list) from `GET /reference/indexes`, and, for the selected index, its recent recorded daily levels (Date, Open, High, Low, Close, Session) from `GET /history/index-daily`.

Rebalancing **does** exist at the API level (`POST /admin/indexes/{id}/rebalance`, via the live `pm-index` bridge) but is deliberately not surfaced as a UI control in this read-only phase — a corporate-action rebalance UI is out of scope. Use `pm-index-admin-cli` for write operations.

Without the stats database the history table shows *"Index history unavailable — the stats database is not running."*

17.10.3 Session Control

The current phase as a badge, and a button for **each transition that is currently legal** — illegal ones are never offered:

From	Offered transitions
`PRE_OPEN`	Opening Auction, Continuous
`OPENING_AUCTION`	Continuous
`CONTINUOUS`	Closing Auction, Closed
`CLOSING_AUCTION`	Closed
`CLOSED`	Pre-Open

Each button confirms first (*"…This affects every participant."*), then posts `/admin/session/transition`. **The REST response is authoritative** for success or rejection; the resulting phase also arrives app-wide a moment later over the live `session.state` broadcast. The engine has the final say and may still reject — most commonly *"Sessions are not enabled on this engine"* or *"Invalid session transition: …"*, surfaced as *"Transition rejected: …"*.

17.10.4 Risk Controls

Read-only views of the resolved static configuration, in three tables:

1. **Risk Levels** — each named level's static and dynamic collar bands, with the default level badged.
2. **Collar Settings (per symbol)** — the *effective* collar for each symbol plus a **Profile** column naming where it came from: `symbol` when the symbol defines its own, otherwise the name of the risk level it inherits from. It also shows each symbol's tick decimals, circuit-breaker level count, and live best bid and ask.
3. **Circuit Breaker Ladder (per symbol)** — level name, price shift %, and halt duration, striped per symbol.

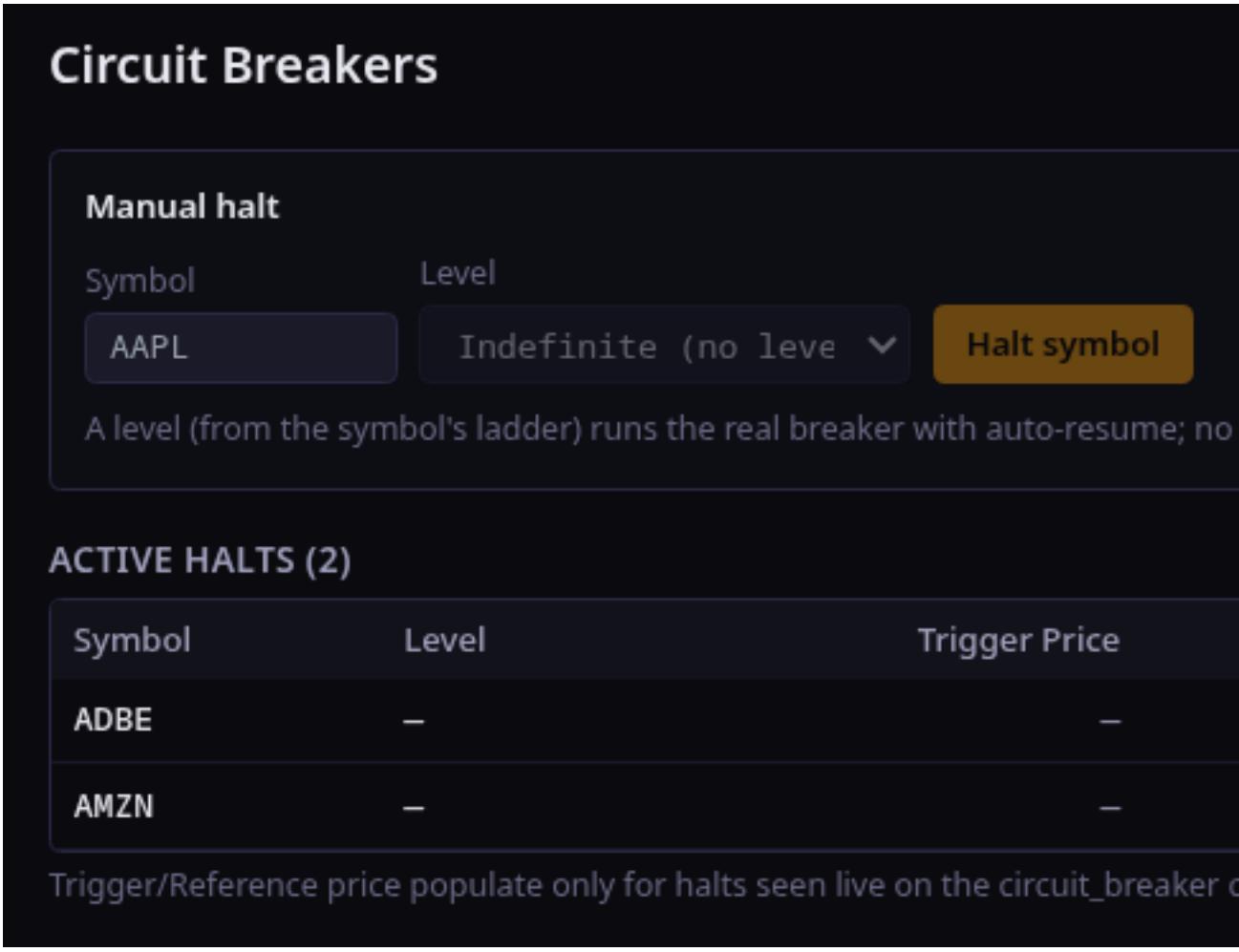


Figure 17.4: Circuit Breakers: the manual-halt form and two active halts.

The two per-symbol tables share one filter and pager: filter by symbol and choose 15, 30 or 50 symbols per page (15 by default).

Nothing here is editable; risk configuration is loaded from `engine_config.yaml`. To list a new symbol, use `pm-new-symbol` (`pm-ipo`) and redeploy. Note the standing rule printed on the page: every halt reopens via a call auction — there is no per-level resumption mode.

17.10.5 Circuit Breakers

Both halts were set by the operator without a level, so Est. Resume reads `indefinite` *and the trigger and reference prices are blank;* ***Clear*** *resumes trading. The Level selector offers only* Indefinite (no level)*: this exchange configures no breaker ladder for the symbol.*

The live operational counterpart to Risk Controls, bootstrapped from `GET /halts` and kept current by the `circuit_breaker` market-data channel.

Manual halt takes a symbol and an optional **Level**, populated from that symbol's *own* configured ladder

(the selector is disabled when the symbol has no ladder). The distinction matters:

- **With a level** — runs the real breaker for that level, including its configured halt duration and **auto-resume**.
- **Without a level** (*"Indefinite (no level)"*) — halts the symbol until an explicit clear.

Either way it is the real breaker, not a simulation, and it confirms first.

Active Halts lists Symbol, Level, Trigger Price, Reference Price, Est. Resume (a timestamp, or `indefinite`), Source, and a **Clear** action which confirms and then resumes trading.

Blank trigger/reference prices are expected

Those two columns populate only for halts your session *observed live* on the `circuit_breaker` channel. Rows restored from the bootstrap snapshot — for instance a halt that fired before you logged in — leave them blank until the symbol halts again. The halt itself is real regardless.

17.10.6 Gateway Management and the Kill Switch

The roster from `GET /admin/gateways` shows Gateway ID, Role, Description and a live connection dot (`Connected` / `Offline`), with an `N / M connected` count and a **Refresh** button in the header.

Kick disconnects a gateway. It always confirms, and the confirmation spells out the side effect: disconnecting a gateway also **cancels all of its active orders and quotes**. It is disabled for gateways that are already offline.

The **Kill Switch** panel below cancels resting orders and quotes at three scopes. All three always confirm regardless of the power-user setting, and all three report what they actually cancelled (*"Kill switch AAPL: 12 orders, 2 quotes cancelled"*).

Scope	Effect
By Symbol	Cancels every resting order and quote for that symbol, across every gateway
By Gateway	Cancels everything belonging to one gateway; the gateway stays connected and may re-submit
Global	Cancels everything for every gateway, reporting the affected gateway count

The **Global** scope additionally requires typing `CONFIRM` before it will run, since it is a full-market emergency stop.

The kill switch cancels; it does not halt

Kill-switch scopes remove exposure. They do **not** stop trading — participants can immediately submit new orders. To stop trading, use a circuit-breaker halt or a session transition.

17.10.7 Monitor Log

The live, filterable tail of cross-gateway activity from the admin monitor feed. Columns: Time (`HH:MM:SS.mmm`), Seq, Type, Order / Gateway, Symbol, Details. The header reports `shown / total` events and a reconciled timestamp, and flags a degraded feed.

Filter by **Event Type** — ALL, ACK, FILL, CANCEL, AMEND, REJECT, EXPIRE, SESSION, CB, ADMIN — and by substring on **Symbol** and **Gateway**. **Export CSV** downloads exactly the rows currently visible under your filters.

A red **GAP** row marks any boundary the feed could not replay after a reconnect. It also marks events the gateway dropped while the feed was overloaded (*"N events dropped by the gateway"*), detected from the per-topic sequence numbers. After such a gap the GUI re-reads the working orders from the gateway, at most once every 10 s, so the order counts and tables recover even though the dropped log rows themselves cannot be replayed. GAP rows are never filtered out, deliberately: a gap is worth noting before trusting the log as complete for that window.

Rows carrying an order id are clickable through to the cross-gateway order drill-down.

Cross-gateway order drill-down

Clicking an order id — here or on the Dashboard — opens a modal showing that order's full audited lifecycle: timestamp, topic, gateway id and a compact payload summary per event.

This reads the **audit trail** (GET /admin/orders/{id}), not the gateway-scoped /history/orders, because only the audit trail can see other gateways' orders. It therefore depends on pm-audit:

- Without it, the modal says *"Audit trail unavailable — pm-audit is not running or its index has not been built."*
- For an unknown id it says *"No audited events for this order."*

17.10.8 Admin capability notes

A few ADMIN controls remain intentionally read-only or scoped, because the backend capability they would need is either not yet exposed or was only recently added. As of this writing:

Capability	Status
Session transition, gateway roster + kick, halts snapshot	Fully live
Circuit-breaker manual trigger with a specific level, and resume/clear	Fully live — the level selector is functional, not disabled
Kill switch — by symbol, by gateway, and global	Fully live
Cross-gateway order drill-down	Live, but requires pm-audit
Symbol add/edit	Read-only — no runtime add-symbol engine command; edit engine_config.yaml and restart pm-engine
Index rebalance	Read-only in this UI — the API endpoint exists, but is intentionally not surfaced here; use pm-index-admin-cli

17.11 Errors and how to read them

Errors in this app fall into five distinct classes. Telling them apart is the single most useful diagnostic skill here — in particular, **class C is not a rejection**, and treating it as one leads to duplicate orders.

17.11.1 Class A — client-side validation

Checked in the browser before anything is sent, so they appear instantly under the relevant field. Nothing reached the gateway.

Message	When it appears
"Symbol required"	No symbol entered (Order Entry only — the Workspace ticket has the symbol locked)
"Quantity must be a positive integer"	Quantity is blank, zero, negative, or not a whole number
"Price required for this order type"	LIMIT, FOK, IOC, or STOP_LIMIT with no price
"Stop price required"	STOP or STOP_LIMIT with no stop price
"Visible qty required"	ICEBERG with no visible quantity
"Visible qty must be less than total qty"	ICEBERG visible quantity total quantity
"Trail offset required"	TRAILING_STOP with no trail offset
"Price required for a LIMIT leg" / "Stop price required for a STOP leg"	OCO or Combo leg missing its required price field
"Market is closed — no orders accepted"	Submitting while the phase is `CLOSED`
"`{TYPE}` orders are not accepted during an auction"	MARKET / FOK / IOC during a call phase
"Price must be a positive number"	Amend dialog, non-numeric or non-positive price
"Quantity must be a positive integer"	Amend dialog, blank / zero / negative / fractional quantity
"Quantity must exceed the `{filled}` already filled — cancel the order instead"	Amend dialog; mirrors the engine's `qty > filled` rule exactly, including the boundary
"No changes to submit"	Amend dialog with nothing edited
"Ask price must be strictly greater than bid price"	New Quote form, crossed or locked quote

17.11.2 Class B — engine rejections

The order reached `pm-engine` and was refused. You get an error toast *"REJECTED: `{reason}`"* and a `REJECT` entry in the Event Center; no row appears in the blotter. The exact reason text is whatever the engine returned; these are the buckets:

Bucket	Typical reason text	Notes
Unknown instrument	`Symbol not configured: XYZ`	The symbol is not in `engine_config.yaml`
Session phase	`Market is closed`	Also enforced client-side
Cannot rest in this phase	`MARKET orders not accepted during OPENING_AUCTION`	MARKET / FOK / IOC only

Bucket	Typical reason text	Notes
TIF window	`ATO orders only accepted during opening auction` (and the ATC equivalent)	The ticket normally prevents this, but a phase change mid-flight can still produce it
Instrument halt	`AAPL is halted — FOK orders rejected during circuit breaker halt`	MARKET / FOK / IOC only. Every other type is accepted and simply rests without matching until the halt clears
Price collar	Collar breach text from the collar validator	A LIMIT / ICEBERG priced too far from the reference price (static band) or the last trade (dynamic band). MARKET carries no price and is exempt
Insufficient liquidity	`Insufficient liquidity`	FOK could not fill in full
Trailing stop setup	`Trailing stop requires STOP= or a prior trade price`	The symbol has never traded and you supplied no stop
Combo leg	`Leg 2 (MSFT) …`	One leg failed; the whole combo is refused
Role	`Quotes are only allowed for MARKET_MAKER participants`	Engine-side role enforcement

Amend and cancel have their own rejection set:

Message	Cause
`Order not found`	The id is unknown, or already evicted
`Cannot amend FILLED order` (or CANCELLED / REJECTED / EXPIRED)	Terminal orders cannot be amended
`Cannot amend {TYPE} orders`	Only LIMIT and ICEBERG are amendable
`Cannot amend an order owned by another gateway` / `Cannot cancel an order owned by another gateway`	Cross-gateway attempt
`Amend requires at least PRICE or QTY`	Empty amend — normally caught locally first
`Amend quantity must be an integer`	Non-integer quantity — normally caught locally first
`New quantity must exceed already-filled quantity`	Normally caught locally first; see the tip under Amending an order
A collar breach	A price amend is collar-checked exactly like a new order

The last three are the engine's wording for rules the Amend dialog also enforces client-side, so in practice you meet the dialog's phrasing rather than these. They are listed because the engine remains the authority — a direct API client, or a race against a fill that changes the filled quantity under you, can still produce them.

OCO cascades are not errors: *"OCO sibling {id} reached {STATUS}"* is the expected cancel reason on the surviving leg.

17.11.3 Class C — timeouts, which are not rejections

A late acknowledgement means the command reached the engine but its acknowledgement did not come back in time. The action may well have succeeded. It can only happen on the actions that wait for the engine's answer. For an order submit the gateway says so explicitly (`202` with `ack_timeout: true`); for the others it is `503 ENGINE_TIMEOUT`. The app never presents these as rejections; each uses neutral "awaiting confirmation" wording:

Action	Message
Order submit	*"{side} {symbol} submitted — awaiting confirmation (check blotter)"*
Session transition	*"Transition submitted — awaiting engine confirmation"*
CB halt	*"Halt submitted for {symbol} — awaiting confirmation"*
CB resume	*"Resume submitted for {symbol} — awaiting confirmation"*
Kill switch	*"Kill switch submitted — awaiting engine confirmation"*

Amend, cancel, group and quote cancels, and Flatten don't wait: the gateway answers at once and the outcome arrives in the blotter. Any error on them, including a `503`, means nothing was sent. Only `ENGINE_TIMEOUT` means "sent, unconfirmed"; a `503 ENGINE_UNAVAILABLE` or `SYMBOLS_NOT_READY` is a plain failure, shown as in class D, and safe to retry.

Do not resubmit. Reconcile instead: check Active Orders (or hit its Refresh), or check the relevant admin screen, and act on what you actually see.

Cancel-Replace has its own wording, because the gateway never sends a replacement for an unconfirmed cancel — see Cancel-Replace.

17.11.4 Class D — gateway and HTTP errors

Anything else that comes back from `pm-api-gwy` is shown as *"{CODE}: {message}"*. The codes you are most likely to meet:

Code	HTTP	Meaning
`AUTH`	401	Missing, malformed or unknown API key
`ENGINE_AUTH`	403	The key is valid but the engine refused the gateway id

Code	HTTP	Meaning
READ_ONLY	403	The key has no gateway_id and cannot write
ROLE_DENIED	403	An /admin/* endpoint called by a non-ADMIN key
ROLE_UNSUPPORTED	—	Client-side: /status reported a role this app cannot use (i.e. READ_ONLY)
VALIDATION	400	Malformed request or bad query parameter
RATE_LIMIT	429	Write rate exceeded
TRANSITION_REJECTED	409	Session transition refused by the engine
ENGINE_TIMEOUT	503	See class C above
ENGINE_UNAVAILABLE	503	The gateway could not hand the command to the engine; nothing was sent, retry
SYMBOLS_NOT_READY	503	The symbol's reference data has not reached the gateway yet; nothing was sent, retry
REPLACE_REJECTED	409	The engine refused a Replace's cancel; see Cancel-Replace
STATS_DB	503	pm-stats unavailable — history endpoints only
AUDIT_INDEX_UNAVAILABLE	503	pm-audit unavailable — admin order drill-down only
UNKNOWN_ORDER	404	No audited events for that order id

17.11.5 Class E — degraded data, not errors

Several panels can legitimately have nothing to show, and say so in plain language rather than failing:

Where	Message	Cause
Market Overview	*"Daily rollup unavailable — Chg %/Volume hidden"*	No stats database; live prices unaffected
Order Detail drawer	*"History unavailable — the stats database is not running. Live events below still update."*	No stats database

Where	Message	Cause
Trade History	*"Could not load fills — is the stats DB available?"*	No stats database
Stats tab	*"No daily statistics for* `{symbol}` *yet"*	No trade has printed this session
Auction tab	*"The engine has not published an indicative uncross for this symbol yet."*	Call phase just started
Company tab	*"The company database is not available on this venue (pm-compdb-cli init builds it)."*	No `companies.db` where `pm-api-gwy` looks for it
Company tab	*"No company profile for* `{symbol}`*."*	The company database has no row for this symbol
Auction badge / panel	`no cross`	A real reading: the book would not cross at any price
Quote Bootstrap	The degraded-leg note	The gateway served `/quotes/legs` from its warm cache
Index Administration	*"Index history unavailable — the stats database is not running."*	No stats database
Admin order drill-down	*"Audit trail unavailable…"*	No `pm-audit`
Connection banner	*"Reconnecting…"* / *"Disconnected…"*	A live socket is down; screens are stale, not wrong

17.12 Typical workflows

17.12.1 A TRADER session

1. **Log in** with a TRADER-role API key; you land on the **Trading Workspace**, already showing a default symbol.
2. **Check the market** — glance at Market Overview or star a few symbols to the Watchlist (which also subscribes them to full depth), then click a row to open Symbol Detail and set the active symbol the ticket and chart follow.
3. **Read the book** — the depth ladder shows live bid/ask levels for the active symbol; clicking a level pre-fills the ticket's price and highlights the matching side.
4. **Place a LIMIT order** — enter quantity and price, leave TIF at DAY, and press Buy or Sell (or `B`/`S`). You get an immediate accepted/rejected answer; an accepted order shows up in the compact blotter as `NEW`.
5. **Watch it rest** — a partial fill flips the status to `PARTIAL`, drops a fill toast, and flashes the Remaining column; press `F4` or switch to the full **Active Orders** screen for more room.
6. **Amend if the market moves** — reduce the order's size at the same price to keep queue priority, or open Replace if you need a genuinely different price or a larger size (accepting the loss of priority).
7. **Cancel or let it fill** — cancel what's left if you've changed your mind, or let it run to `FILLED`. Either way it goes terminal: the row stays in the blotter with a terminal pill and greyed buttons until the

gateway evicts it, and its full story stays available in the Order Detail drawer and Trade History.

8. **Check the result** — **Positions** (F3) shows your updated net position and last price; **Flatten** it with one click if you want to close out (MARKET order, only available during continuous trading). **Trade History** has the durable fill record, filterable by symbol/side/date.

17.12.2 A MARKET_MAKER session

1. **Log in** with a MARKET_MAKER-role API key; you land on **Quote Management**, a card per configured symbol.
2. **Enter a two-sided quote** — press F2 (or click New Quote on a card) to open the quote form for the active symbol: quote id, bid price/qty, ask price/qty, and TIF. Watch the spread indicator as you type; the form won't let the ask price sit at or below the bid.
3. **Watch the legs** — each card shows a fill-progress bar per side as the market trades against your quote.
4. **React to a fill** — when a leg fills, the quote goes inactive on that side and you get a toast with a **Re-quote** button; clicking it re-opens the New Quote form pre-filled with your previous price and size so you can decide whether to hold the same market or adjust before resubmitting.
5. **Cancel and replace as needed** — cancel a quote from its card (always confirms; removes both legs at once, and a symbol carries one active quote per gateway) and submit a fresh one when your view of the market changes.
6. **Diagnose if something looks off** — **Quote Bootstrap & Legs** shows the raw active-quote snapshot alongside the per-leg table, with **Resync** and a note flagging rows that came back in the degraded quote-level shape.
7. **Check the result** — the same **Positions** screen used by TRADER shows your net exposure across symbols, with the same Flatten / Flatten All available during continuous trading.

17.12.3 An ADMIN session

1. **Log in** with an ADMIN-role API key; you land on the **System Dashboard** — session phase, active orders across every gateway, connected-gateway count, and active circuit-breaker halts at a glance, plus a live cross-gateway events feed.
2. **Monitor for trouble** — the per-symbol summary table and the events feed are where an unusual halt, a gateway drop, or a burst of rejects would first show up; click any event row carrying an order id for the full audited lifecycle.
3. **Respond to a circuit-breaker halt** — on **Circuit Breakers**, active halts show trigger price, reference price, estimated resume time and source. Clear one with the **Clear** button on its row. To halt a symbol yourself, pick it, then either pick a level from its configured ladder (the real breaker, with auto-resume) or leave the level blank for an indefinite halt, and confirm.
4. **Use the kill switch in an emergency** — on **Gateway Management**, the **Kill Switch** panel cancels resting orders and quotes by symbol, by gateway, or globally. All three scopes always confirm, and the global scope additionally requires typing CONFIRM. Remember it cancels exposure but does *not* stop trading.
5. **Manage the session** — on **Session Control**, only the phase transitions that are currently legal are offered as buttons; confirm to transition, and watch the resulting phase arrive over the live session broadcast a moment later.
6. **Kick a misbehaving gateway** — from the gateway roster, **Kick** disconnects it and, as a side effect, cancels its resting orders and quotes — useful if a bot or a stuck session needs to be forcibly cleared without waiting for a broader halt.
7. **Review afterward** — **Monitor Log** gives a filterable, exportable tail of everything that happened

across every gateway, with the same order-lifecycle drill-down used on the Dashboard; a GAP row marks any stretch the feed couldn't replay after a reconnect, worth noting before trusting the log as complete for that window.

17.13 Keyboard shortcuts and power-user mode

Press **Ctrl+/** to open the help drawer, or **?** to open the shortcut reference card directly (outside of a text input). Both render the same table as below, from the same source, so they never drift apart.

Keys	Scope	Action
F1	Global	Focus the order ticket (TRADER)
F2	MARKET_MAKER	New quote form for the active symbol
F3	Global	Toggle the position panel
F4	TRADER	Toggle the order blotter
B	Order ticket	Submit BUY with the ticket's parameters
S	Order ticket	Submit SELL with the ticket's parameters
Ctrl+K	Global	Open the command palette (symbol / action search)
Ctrl+.	Global	Toggle the Notification / Event Center
Ctrl+L	Global	Toggle the Watchlist panel
Shift+F	Position row	Flatten the selected position (MARKET close)
Ctrl+Shift+F	Global	Flatten All (always confirms)
Escape	Global	Close the open modal / panel / drawer
Ctrl+Enter	Form focus	Submit the focused form (OCO / Combo)
Ctrl+/	Global	Toggle the help drawer
?	Not in an input	Open this keyboard shortcut reference
Delete / Backspace	Blotter row	Cancel the selected order
↑ / ↓	Blotter	Navigate rows
Ctrl+A	Blotter	Select all visible rows
Enter	Blotter row	Open the Order Detail drawer

A few behavioural notes:

- `B` and `S` are **ignored while a form field has focus**, so typing a symbol containing those letters never fires an order.
- `Ctrl+K`, `Ctrl+.`, `Ctrl+L`, `F1`–`F4` and `Ctrl+/` work even with a field focused. `Ctrl+L` and `F3` collide with browser shortcuts; the app wins when it has focus.
- `Ctrl+Shift+F` and the palette's "Flatten All" both **navigate to Positions**, where the always-confirm dialog lives, rather than firing blind from wherever you were.
- `Escape` inside the order ticket also clears the ticket's error messages and blurs the focused field.
- The five **blotter** keys (`↑`, `↓`, `Ctrl+A`, `Enter`, `Delete`) need a row to have focus first — click one, or Tab into the table. `↑`/`↓` move focus without changing the selection, and `Ctrl+A` selects only *cancellable* rows; see Selecting rows and moving around.

17.13.1 Power-user mode

Power-user mode is the **Confirm cancellations** toggle in the settings popover, on by default.

With it **on** (default)	With it **off** (power-user)
Single-order cancel → confirmation dialog	Single-order cancel → fires immediately, undo toast
Per-row Flatten → confirmation dialog	Per-row Flatten → fires immediately, undo toast (best-effort: it cancels the MARKET order if it hasn't filled)

These actions **always confirm regardless of the setting**, because none of them is reversible with an undo:

- Bulk cancel (multiple selected orders)
- OCO / Combo group cancel
- **Quote cancel** — it removes two resting legs at once
- Flatten All
- Session transition
- Circuit-breaker halt and clear
- Gateway Kick
- Kill switch, all three scopes — and Global additionally requires typing `CONFIRM`

17.14 Settings

The **settings** popover (gear icon, top bar) holds three settings:

Setting	Choices	Default	Kept after a reload?
Confirm cancellations	on / off	on	No
Font size	XS, S, M, L, XL, XXL	M	Yes, per browser
Symbol picker	Auto, Dropdown, Search, plus an **Auto threshold**	Auto, threshold 20	No

Confirm cancellations is the power-user toggle described above.

17.14.1 Font size

Scales the whole application — text, tables, panels and charts together — from XS (the most compact) to XXL, for a large monitor, a projector, or a reader who wants bigger type. M is the default. The choice is stored in this browser's `localStorage`, so it survives a reload and a logout, but another browser starts at M again.

17.14.2 Symbol picker

Chooses the widget the Trading Workspace uses to select its active symbol:

- **Dropdown** — a plain list of every symbol. Quickest with a handful of symbols.
- **Search** — a text box that narrows the list as you type: any symbol *containing* what you typed matches. ↑/↓ move through the matches, **Enter** picks one, **Esc** closes the list. Better with dozens or hundreds of symbols, where a dropdown becomes a long scroll.
- **Auto** (default) — Dropdown while the exchange lists fewer symbols than the **Auto threshold** (20 by default), Search once it lists that many or more. A classroom exchange with three books gets a dropdown and one with thirty gets a search box, with nothing to set.

Unlike the font size, this choice and the threshold are held in memory only and return to Auto / 20 when the page is reloaded.

17.15 Configuration reference

The app reads its configuration from Vite environment variables at build time. Copy `web-apps/trader-gui/.env.e` to `.env` and adjust.

Variable	Default	Purpose
`VITE_API_BASE`	*(empty)*	Base URL for the `pm-api-gwy` REST API. Empty relies on the dev proxy / same-origin reverse proxy.
`VITE_WS_BASE`	*(empty)*	Base URL for WebSocket connections. Empty resolves against the page origin (`http`→`ws`, `https`→`wss`).
`VITE_APP_TITLE`	`EduMatcher Trading`	Title on the login screen, shown as `<title> · pm-trading-ui`. The browser tab title is fixed in `apps/web/index.html`, and the top bar always shows `EduMatcher pm-trader`.
`VITE_MAX_OVERVIEW_SYMBOLS`	`250`	Cap on the broad book/trades subscription (Market Overview).
`VITE_MAX_FOCUS_SYMBOLS`	`25`	Cap on the focused per-symbol depth/auction subscription set (active symbol + watchlist).

Variable	Default	Purpose
VITE_CHART_HISTORY_TICKS	1000	Historical prints fetched for an intraday chart.
VITE_FLASH_DURATION_MS	500	Price flash-cell animation duration.
VITE_MARKET_THROTTLE_MS	250	How often Market Overview re-derives its rows from the book store — bounds re-render work independently of symbol count.
VITE_WS_RECONNECT_MAX_DELAY	30000	WebSocket reconnect backoff cap, in milliseconds.
VITE_NOTIFICATION_BUFFER	500	Maximum entries retained in the Event Center.

A mistyped integer value falls back to its default rather than becoming NaN and silently disabling the cap it was meant to set.

For the production static server (pm-trading-ui-serve), the relevant variables are plain environment variables rather than VITE_* build-time values:

Variable	Default	Purpose
HOST	0.0.0.0	Bind address.
PORT	8093	Listen port.
STATIC_DIR	apps/web/dist/	Path to the built SPA.
API_PROXY_TARGET	*(unset — /api/* returns 503)*	Forward /api/* to this URL, e.g. http://localhost:8080.

This table is generated from the same declaration the server reads at startup, so npm run serve -- --help prints exactly these four options and their real defaults — including the absolute STATIC_DIR path resolved for your checkout. Treat that output as authoritative if it ever disagrees with this page.

Both container paths set the same pm-trading-ui-serve variables, plus a host-side port mapping — but they differ in exactly one place, which is the thing worth knowing:

Variable	In the whole stack (compose.guis.yaml)	In this app alone (web-apps/trader-gui/docker-compose
TRADER_GUI_PORT	8093 — host port; change it if 8093 is taken	8093 — same
HOST / PORT	0.0.0.0 / 8093 — bind inside the container	0.0.0.0 / 8093 — same

Variable	In the whole stack (compose.guis.yaml)	In this app alone (web-apps/trader-gui/docker-compose
API_PROXY_TARGET	http://edumatcher:8080 — the backend's service name on the shared network; nothing to configure	http://host.docker.internal:8080 — reaches back out to your host; **set it explicitly on Podman or Linux Docker**

That single row is why the whole-stack path needs no addresses: both containers are on one Compose network, so the exchange has a name (edumatcher) that simply resolves. Running this app on its own puts it outside that network, and it has to be told how to get back in.

17.16 Troubleshooting

Symptom	Likely cause	What to check
Login fails with "Invalid API key"	The key is missing, mistyped, or not present in engine_config.yaml	Check api_gateways.<name>.credentials and confirm the key you pasted matches exactly
Login fails with "ROLE_UNSUPPORTED: Unsupported role: READ_ONLY"	The key is a read-only credential (gateway_id: null)	Use a key bound to a gateway_id; read-only keys are for observer tools like TapeDeck
Login fails with "ENGINE_AUTH: …"	The key is valid but pm-engine refused its gateway id	Confirm the gateway_id exists under participants in engine_config.yaml. "Gateway already connected" means another client holds that id — a bot, a console, or pm-api-gwy itself, which keeps every id it has authenticated for as long as it runs. An ADMIN can free it with **Kick** on Gateway Management (this cancels the gateway's orders and quotes)
Login fails with "Gateway reached, but the engine did not answer. Is pm-engine running?"	pm-api-gwy is up but cannot reach the matching engine	Confirm pm-engine is running and the gateway's connection to it is healthy
"Cannot reach the API gateway. Is pm-api-gwy running?"	The browser's fetch() to pm-api-gwy failed outright	Confirm pm-api-gwy is running and reachable at VITE_API_BASE (dev) or behind your reverse proxy (production)

Symptom	Likely cause	What to check
Connection banner shows "Reconnecting…" or "Disconnected" after a successful login	The events/market-data WebSocket dropped	Confirm `pm-api-gwy` is still running; the banner clears automatically once the socket reconnects, and `orders.snapshot` resyncs the blotter
Change % and Volume columns are blank on Market Overview	The daily stats rollup is unavailable	Check that the stats database used by `pm-api-gwy`'s `/history/daily` endpoint is reachable; live prices are unaffected
Trade History, the chart's history, or the Order Detail timeline is empty	Same cause — no `pm-stats`	Live fills and live events still appear; only the durable record is missing
The blotter looks stale or disagrees with the engine	A missed event or a long reconnect	Press **Refresh** on Active Orders for an explicit `GET /orders` reconcile
An order I submitted isn't in the blotter and I got no rejection	The ack timed out (`ack_timeout`)	Do **not** resubmit — press Refresh and look again; see class C errors
MARKET / FOK / IOC tabs are disabled	An auction phase is running	Expected — those three cannot rest, so the engine refuses them during a call phase
Flatten and Flatten All are disabled	The phase is not `CONTINUOUS`	Expected — flatten submits MARKET orders
A quote was rejected with "Quotes are only allowed for MARKET_MAKER participants"	The key's gateway is not a market maker	Use a `MARKET_MAKER` credential
Quote Bootstrap legs show blank per-leg columns	The gateway served `/quotes/legs` from its warm cache	Expected; use the Active Quotes table above for authoritative per-side price/qty, or press **Resync**
Circuit-breaker Level selector is empty for a symbol	That symbol has no configured CB ladder	Check `circuit_breaker.levels` for the symbol in `engine_config.yaml`; an empty ladder is valid, just leaves only the indefinite-halt option
Active halts show blank Trigger/Reference price	The halt was restored from the bootstrap, not observed live	Expected; the halt is real, the prices populate only for halts seen on the live `circuit_breaker` channel

Symptom	Likely cause	What to check
Admin order drill-down says "Audit trail unavailable"	`pm-audit` is not running or has not built its index	Start `pm-audit`; the Monitor Log tail itself is unaffected
A red `GAP` row in the Monitor Log	The monitor feed could not replay a stretch after a reconnect, or the gateway dropped events under load	Expected after a disconnect or a load burst; treat the log as incomplete for that window (working orders are resynced automatically)
Reloading the page returns to the login screen	Expected — the API key is held in memory only, never in browser storage	Re-enter the key; this is by design for a localhost classroom system

17.17 Related documentation

- API Gateway — the REST + WebSocket surface the Trading GUI is built entirely on
- Order Types — the engine-level reference for every type the ticket offers, including amendment priority rules
- Combo Orders — the engine's multi-leg semantics
- Session Scheduling — the phases Session Control transitions between
- Market Making — quoting obligations and the engine's quote model
- Risk Controls — the collar and circuit-breaker concepts shown read-only in the ADMIN Risk Controls screen
- Index Administration (`pm-index-admin-cli`) — the write path for index rebalancing, which this UI deliberately does not expose
- Audit — the trail behind the ADMIN cross-gateway order drill-down
- Trader Information Terminal ("TapeDeck") — the sibling read-only market display; use it for a classroom wallboard or observer view instead of the Trading GUI
- `docs-design/EduMatcher-Trading-GUI.md` — the full design document, including the backend capability matrix (§6) and complete screen specifications (repository checkout only)
- `web-apps/trader-gui/README.md` — the implementation's own record of setup, environment variables, and phase-by-phase implementation status (repository checkout only)

CHAPTER 18

Order Book Viewer (pm-book)

Learning objectives

After reading this page you will understand:

- What the order book viewer shows, and how it relates to `pm-viewer`
- Which services must be running, and why the read-only API key is mandatory for this application
- How to start it with the whole stack, on its own in a container, on a separate display server, or as a local development server
- How the Compose setup wires it to the exchange
- What every field of the statistics header, the two ladders and the trade tape means
- How to read the connection states and the footer without mistaking a quiet market for a broken one
- Which environment variables an operator is most likely to adjust

18.1 Overview

`pm-book` is the web companion to the `pm-viewer` terminal application: one symbol's full order book — every price level on both sides — the session statistics above it and a trade tape beside it, in a browser. Its source code lives in `web-apps/book-gui/`.

Like TapeDeck it is read-only. There is no order entry, no login screen and no write path from the browser into the exchange. Use it on a classroom projector while students trade against a book, beside the Trading GUI to watch the effect of your own orders, or on any machine where `pm-viewer` would need a terminal and direct access to the engine's message bus.

Everything `pm-viewer` shows is here, field for field: last price and change, bid, ask and spread, open/high/low/close previous close, range, volume, and the basis the change is measured against. What the browser adds is symbol switching by URL, a dark and a light theme, font scaling, and a layout that fills whatever screen it is shown on.

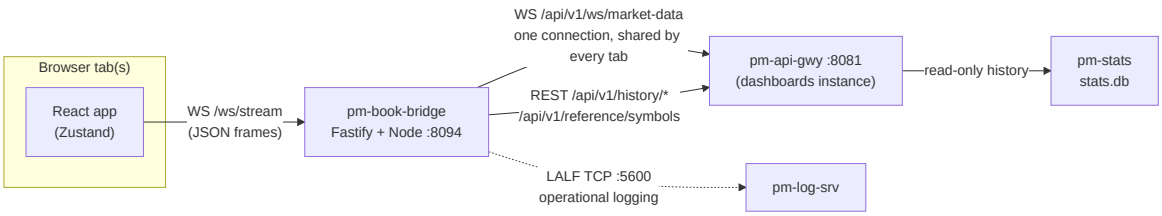
18.1.1 What starts when you run it

Running the viewer starts one server process, `pm-book-bridge`, which also serves the web page. The bridge reads its market data from the API Gateway — it never talks to the engine or to `pm-stats` directly:

From pm-api-gwy	Used for
WS /api/v1/ws/market-data	The live book and every new trade of the symbols being watched
GET /api/v1/reference/symbols	The symbol list in the picker, and each symbol's number of price decimals
GET /api/v1/session	The exchange session timezone and today's trading date
GET /api/v1/history/trades	The session's earlier trades, so the statistics and tape are complete when you open a book
GET /api/v1/history/daily	The previous close

The one exception is the session phase in the top bar, which the bridge reads from `pm-md-gwy` over CALF (TCP 5570) — the same source as TapeDeck, and one that needs no key.

The bridge holds exactly one upstream WebSocket however many browser tabs are open, and subscribes to a symbol only while at least one tab is watching it. The read-only API key stays in the bridge; the browser never sees it.



18.2 Prerequisites

Requirement	Notes
pm-api-gwy (API Gateway), dashboards instance, with a **read-only API key**	Required for everything. The key is the credential with `gateway_id: null`; it is issued on the `dashboards` instance (port 8081 in the bundled configurations), not on `desk` (8080). Without the key the viewer has nothing to show.
pm-stats (Statistics)	Strongly recommended. The history endpoints read its database. Without it the viewer still works, but the statistics cover only the trades seen since the book was opened, and the header says so.
pm-log-srv (Centralized Log Server) — optional	Used only for the bridge's own operational logs. If it is unavailable the bridge logs to stdout, or to a local failover file.

Requirement	Notes
Podman 4 or **Docker 24** with a Compose plugin	Needed for the container run paths.
Node.js 20 and **npm 10**	Needed only for local development without a container.

The key is mandatory — unlike TapeDeck

TapeDeck gets its live prices from `pm-md-gwy` and uses the API key only for history, so it still ticks without one. The order book viewer reads its live data through `pm-api-gwy` too. Without a key the bridge starts, logs `PM_BOOK_API_KEY is unset — no market data or history can be read`, and never connects upstream. The app's own `make up` therefore refuses to start without one.
Mandatory does not mean you have to find it: every bundled configuration already contains the key, and the local start paths look it up for you — see Where the read-only API key comes from.

18.3 Running the application

There are four ways to run the viewer. They differ only in how the bridge learns two things: **where `pm-api-gwy` is**, and **the read-only API key**.

Way	Where the exchange runs	Who supplies the address and the key
The whole stack	In the same Compose project	Found automatically
This app alone, in a container	On this host, in a VM, or elsewhere	You, or `make up` from the deployed configuration
On a separate display server	On another machine	You
Local development	Anywhere reachable	You, or `make dev-env`

You only ever supply the key yourself when the exchange runs on another machine; everywhere else it is looked up for you, as described in Where the read-only API key comes from.

18.3.1 The whole stack (recommended)

This path requires nothing of you: it starts the exchange and every web application together, reads the read-only key out of the deployed configuration, and points the viewer at the backend by service name.

A released install (the one-line installer):

```
curl -fsSL https://raw.githubusercontent.com/johan162/EduMatcher/main/deployment/curl/install.sh | bash
cd ~/.edumatcher
./edumatcher.sh start
```

From a source checkout:

```
cd deployment/docker
make up-all
```

Then open **http://localhost:8094**. It opens on the first listed symbol; press `s` to pick another. `./edumatcher.sh urls` (released install) or the summary `make up-all` prints lists every application's address.

An install from before the order book viewer existed

`./edumatcher.sh update` pulls new images but does not re-download `compose.yaml`, so an older install does not gain the viewer from an update. Re-run the one-line installer: it keeps your `.env` and `data/` and fetches the current `compose.yaml` and `edumatcher.sh`. See Installation.
Useful commands on this path:

Task	Released install	Source checkout (`deployment/docker`)
Follow the viewer's log	`./edumatcher.sh logs book-gui`	`podman logs -f edumatcher-book-gui`
Rebuild only this image and restart	—	`make up-all BUILD=1 GUI=book-gui`
Move it to another host port	`BOOK_GUI_PORT=8100` in `.env`	`BOOK_GUI_PORT=8100` in `.env`
Stop everything	`./edumatcher.sh stop`	`make down-all`

Use `docker` in place of `podman` if that is your runtime. After changing the engine configuration (`make up-all CONFIG=<other>`), start the stack again rather than only the viewer: every bundled configuration carries a different read-only key, and the key is resolved at start time.

18.3.2 This app alone, in a container

Use this when the exchange is already running outside Compose — as processes on your host, in a VM, or on another machine. From `web-apps/book-gui/`:

```
export EDUMATCHER_DATA_DIR=~/.local/share/edumatcher
make up
```

Then open **http://localhost:8094**. `make logs` follows the bridge log, `make ps` shows the container, and `make down` stops it.

How `make up` finds the key. An explicit `PM_BOOK_API_KEY` always wins:

```
make up PM_BOOK_API_KEY=key-readonly-...
```

Otherwise `make up` reads it from the deployed configuration `$EDUMATCHER_DATA_DIR/ref_data/engine_config.json` — the same file the exchange's own processes read — picking the first credential with `gateway_id: null` (the `dashboards` instance is preferred when several carry one). It prints which instance and port the key came from, and warns when `API_GATEWAY_URL` points at a different port. With neither a key nor a deployed configuration to read it from, `make up` stops with an explanation rather than starting a viewer that can show nothing.

The address. The app's `docker-compose.yml` defaults `API_GATEWAY_URL`, `CALF_HOST` and `LOG_SRV_HOST` to `host.docker.internal`, which is right on Docker Desktop. On bare Linux Docker, `make up` adds `docker-compose.li` which maps that name to the host. On Podman, point them at `host.containers.internal`:

```
export API_GATEWAY_URL=http://host.containers.internal:8081
export CALF_HOST=host.containers.internal
export LOG_SRV_HOST=host.containers.internal
make up
```

The key is valid on 8081, not 8080

`API_GATEWAY_URL` must point at the `dashboards` instance, where the read-only credential is issued. `desk` on 8080 rejects it; the bridge then logs `refused PM_BOOK_API_KEY` and the top bar stays on `RECONNECTING`. Keep the port when you change the host.
If port 8094 is taken, move the *host* side of it. Pick something outside 8090–8095, which the other applications use:

```
BOOK_GUI_PORT=8100 make up
```

All the web applications against one backend

`web-apps/Makefile` starts the log console, TapeDeck, the Trading GUI and the order book viewer together, each from its own directory, and points them all at one backend address:

```
cd web-apps
make up VM_BACKEND_IP=192.168.64.10 PM_BOOK_API_KEY=key-readonly-...
make up-book VM_BACKEND_IP=192.168.64.10 PM_BOOK_API_KEY=key-readonly-...   # only this one
make logs-book
make down-book
```

`VM_BACKEND_IP` sets `API_GATEWAY_URL=http://<ip>:8081`, `CALF_HOST=<ip>` and `LOG_SRV_HOST=<ip>` for the viewer. Pass the key explicitly when the backend runs in a VM: the automatic look-up reads the deployed configuration from *this* host's data directory.

Alternative: direct Compose commands

```
PM_BOOK_API_KEY='key-readonly-...' docker compose up --build -d
docker compose logs -f book-gui
docker compose down
```

With Podman, use `podman-compose` for the same commands.

18.3.3 Running on a separate display server

The viewer does not have to run on the machine that runs the exchange. A common classroom set-up has the exchange on one server and the viewer on the machine driving the projector. Nothing changes on the exchange side: the bridge is just another read-only `pm-api-gwy` client.

Exchange-side item	What to check
`pm-api-gwy`	It binds `0.0.0.0` by default. The display server must reach the `dashboards` instance — HTTP and WebSocket on TCP `8081` in the bundled configurations.
API key	The read-only credential (`gateway_id: null`) of that instance. It never leaves the display server's bridge.

Exchange-side item	What to check
`pm-stats`	Must be running on the exchange server for complete statistics; the viewer reaches it only through `pm-api-gwy`.
`pm-md-gwy`	For the session phase in the top bar. Make TCP `5570` reachable and set `CALF_HOST`; without it the phase shows `Unknown`.
`pm-log-srv`	Optional. Make TCP `5600` reachable and set `LOG_SRV_HOST`, or set `LOG_SRV_ENABLED=false`.
Firewall	Open `8081` and `5570` (and optionally `5600`) from the display server to the exchange. Browsers need only the display server's `8094`.

On the display server, use a prepared image rather than building from source:

```
# Option A: the released image
podman pull ghcr.io/johan162/edumatcher-book-gui:<VERSION>

# Option B: an image tarball made with `make cdist` in web-apps/book-gui
podman load --input edumatcher-book-gui-<VERSION>.tar.xz
```

Then run it, pointing it at the exchange server:

```
mkdir -p logs

podman run -d --name book-gui \
  --restart unless-stopped \
  -p 8094:8094 \
  -v "$PWD/logs:/app/logs" \
  -e API_GATEWAY_URL=http://exchange.example.org:8081 \
  -e PM_BOOK_API_KEY='key-readonly-...' \
  -e LOG_SRV_ENABLED=false \
  ghcr.io/johan162/edumatcher-book-gui:<VERSION>
```

Use the image name you pulled or loaded. For centralized logging, replace `LOG_SRV_ENABLED=false` with `-e LOG_SRV_HOST=exchange.example.org`. Then open **http://display-server.example.org:8094**.

If the page loads but the top bar shows `RECONNECTING`, the display server serves the UI but cannot reach or authenticate with `pm-api-gwy`; the bridge log says which.

18.3.4 Local development

From `web-apps/book-gui/`:

```
make install                                  # once, and after a dependency change
PM_BOOK_API_KEY=key-readonly-... make dev     # Vite on :8194, bridge on :5194
```

Open **http://localhost:8194**. The Vite dev server proxies `/api` and `/ws` to the bridge on `127.0.0.1:5194`, so the browser sees a single origin.

The bridge's connect targets default to `127.0.0.1`, which is exactly where the container stack publishes its

ports, so `make up-all` in `deployment/docker` plus `make dev` here needs nothing beyond the key. `make dev-env` prints it, together with the matching `API_GATEWAY_URL`:

```
eval "$(make -s -C ../../deployment/docker dev-env GUI=book-gui)"
make dev
```

Keep dev-env out of the shell you start containers from

`eval "$(make dev-env …)"` exports `API_GATEWAY_URL=http://127.0.0.1:8081` and `PM_BOOK_API_KEY` into your shell, and they stay there. That is right for `make dev`, whose bridge runs on your own machine. It is wrong for a **container** started later from the same shell: Compose hands the exported `API_GATEWAY_URL` to the container, where `127.0.0.1` is the container itself. The app's own `make up` in `web-apps/book-gui` — or `make up` in `web-apps/` without `VM_BACKEND_IP` — then starts a bridge that cannot reach `pm-api-gwy`: the top bar stays on `RECONNECTING`, and the bridge log shows connections to `127.0.0.1:8081` being refused. The exported key also takes precedence over `make up`'s own look-up, and is stale once the exchange runs another configuration.

`make up-all` and `./edumatcher.sh start` are not affected: whenever they find the read-only credential they set both values themselves.

Either load the variables for the development server only, in a subshell:

```
(eval "$(make -s -C ../../deployment/docker dev-env GUI=book-gui)"; make dev)
```

or clear them before starting a container from that shell:

```
unset API_GATEWAY_URL PM_BOOK_API_KEY
```

The containerised viewer keeps running on 8094 at the same time, which makes it a convenient reference for your change on 8194. The full inner loop is described in The Development Loop.

18.3.5 Make targets

Target	Description
`make install`	Install the npm workspace from the lockfile
`make dev` / `dev-web` / `dev-bridge`	Bridge and Vite together / only Vite / only the bridge
`make build` / `build-debug`	Production build / frontend with sourcemaps
`make typecheck` / `lint` / `format`	Type-check every workspace / the same / Prettier
`make test`	The Vitest suite: the bridge end to end against a fake `pm-api-gwy`, the web app against a fake bridge, and unit tests
`make cbuild`	Build the container image without starting it
`make up` / `down` / `restart` / `logs` / `ps`	Container lifecycle
`make cdist`	Build the image and export it as `dist/edumatcher-book-gui-<VERSION>.tar.xz`
`make clean` / `distclean`	Remove build output / also `node_modules` and the lockfile

The version in the top bar comes from `apps/web/src/version.json`, which `scripts/mkbld.sh` writes for every web application at release time.

18.3.6 Where the read-only API key comes from

The viewer gets its key exactly the way TapeDeck does, and from the same place — the two applications use the **same** key, under different variable names.

It is part of the configuration. `pm-config-gen`'s `--api-gateway-readonly-key` option adds a credential with `gateway_id: null` and a randomly generated key to the `dashboards` API gateway instance (port 8081). Every bundled example configuration was generated with that option, so each carries one, and each a different one:

```
api_gateways:
  dashboards:
    port: 8081
    credentials:
    - api_key: key-readonly-fq9m76
      gateway_id: null
      description: Generated read-only market-data key
```

Deploying a configuration copies the key into `<data dir>/ref_data/engine_config.json`.

Every local start path looks it up and passes it to the bridge as `PM_BOOK_API_KEY`:

Start path	How it finds the key	With no key to be found
`make up-all` / `./edumatcher.sh start`	Starts the exchange, waits for its configuration to be deployed, reads the key inside the exchange container, and exports it as both `PM_BOOK_API_KEY` and `PM_TERMINAL_API_KEY`, with `API_GATEWAY_URL=http://edumatcher:<port>`; then starts the web applications	Starts anyway and warns; the viewer has nothing to show
`make up` in `web-apps/book-gui`	Reads `$EDUMATCHER_DATA_DIR/ref_data/engine_config.json` on this host. A `PM_BOOK_API_KEY` you set yourself always wins	**Stops** with an explanation
`make dev-env GUI=book-gui`	Reads the key from the running exchange container and prints it as an `export` line for `make dev`	Prints a warning

Each picks the first enabled instance, in name order, with a `gateway_id: null` credential — `dashboards` before `desk` — and takes that instance's port with the key.

When you do pass it yourself. The look-up reads a configuration file on the machine it runs on. When the exchange runs somewhere else — `web-apps/Makefile` with `VM_BACKEND_IP`, or a separate display server — there is no file to read, so you copy the key from that exchange's `engine_config.yaml` (the `gateway_id: null` credential under `dashboards`) and pass it as `PM_BOOK_API_KEY`.

A configuration without one. Add a credential with `gateway_id: null` to the `dashboards` instance — in the Configuration GUI's gateway settings, by hand, or by regenerating with `--api-gateway-readonly-key` — and redeploy. Every configuration has its own key, so after switching configuration restart through a

path that looks it up again; a key copied from another configuration is refused, and the viewer stays on RECONNECTING.

The key reads only public data — reference data, the market-data stream and history — and cannot place or see orders. It stays in the bridge and never reaches the browser.

18.4 The Compose setup

The viewer appears in three compose files. The service is the same in each; what differs is how it finds the exchange.

File	Used by	API_GATEWAY_URL default	Image
deployment/curl/compose.yaml	The one-line installer, edumatcher.sh	http://edumatcher:8081	ghcr.io/johan162/edumatche
deployment/docker/compose.yaml	make up-all	http://edumatcher:8081	edumatcher-book-gui:latest built from web-apps/book-gui
web-apps/book-gui/docker-compose.yml	The app's own make up	http://host.docker.internal:8081	edumatcher-book-gui:latest built locally

In the two whole-stack files the viewer shares one Compose network with the exchange container, which answers to the hostname edumatcher. The source-checkout service looks like this:

```
book-gui:
  image: edumatcher-book-gui:latest
  container_name: ${CONTAINER_NAME:-edumatcher}-book-gui
  depends_on:
    - edumatcher
  restart: unless-stopped
  ports:
    - "${BIND_ADDR:-127.0.0.1}:${BOOK_GUI_PORT:-8094}:8094"
  environment:
    HOST: "0.0.0.0"
    PORT: "8094"
    CORS_ORIGIN: "${CORS_ORIGIN:-*}"
    MAX_WS_CLIENTS: "${MAX_WS_CLIENTS:-200}"
    API_GATEWAY_URL: "${API_GATEWAY_URL:-http://edumatcher:8081}"
    PM_BOOK_API_KEY: "${PM_BOOK_API_KEY:-}"
    LOG_SRV_ENABLED: "${LOG_SRV_ENABLED:-true}"
    CALF_HOST: "edumatcher"
    LOG_SRV_HOST: "edumatcher"
    LOG_SRV_PORT: "5600"
    LOG_FAILOVER_DIR: "/app/logs"
  volumes:
    - book-gui-logs:/app/logs
```

Points worth knowing:

- **PM_BOOK_API_KEY is empty in the file on purpose.** make up-all and ./edumatcher.sh start start the exchange first, read the key from the deployed configuration inside it, and export it for this service —

the same key TapeDeck gets as `PM_TERMINAL_API_KEY`. See The read-only API key.

- **`BIND_ADDR`** (default `127.0.0.1`) decides who can open the page. Set it to `0.0.0.0` in `.env` to serve the viewer to other machines, and consider restricting `CORS_ORIGIN` when you do.
- **There is no data volume.** The bridge keeps each watched symbol's session in memory and rebuilds it from `pm-api-gwy` on restart. The one volume, `book-gui-logs` (`./logs` for the app's own compose file), holds the log failover file, written only when `pm-log-srv` becomes unreachable.
- **A health check** (in the app's own compose file) polls `http://127.0.0.1:8094/api/bridge/status` every 30 seconds.
- **Image build arguments** `HTTP_PROXY`, `HTTPS_PROXY`, `NO_PROXY` and `NPM_STRICT_SSL` are passed through for builds behind a corporate proxy, as for the other web applications.

18.5 A tour of the interface

The top bar, the statistics header, the BIDS and ASKS ladders with depth bars scaled to the largest level on screen, the trade tape coloured by tick, and the footer: `upstream ACTIVE`, how many levels of each side are shown, and the age of the last book update.

The page has four parts, top to bottom: the top bar, the statistics header, the three panels **BIDS**, **ASKS** and **TRADES**, and the footer.

18.5.1 Top bar

- **`EduMatcher pm-book v<version>`** on the left, as in every EduMatcher web application.
- **The symbol picker** — the current symbol with a drop-down arrow. Click it, or press **`s`** or **`F1`** anywhere on the page.
- **The session indicator** — the exchange session phase as a coloured badge, the next phase and the time left to it (`Continuous → Closing Auction in 03:08`), and the clock in the exchange session timezone, exactly as the Trading GUI shows it. The countdown appears only while a transition is scheduled; the badge reads `Unknown` until `pm-md-gwy` has reported the phase, and whenever it cannot be reached.
- **Theme** (sun/moon): switches between the dark and the light palette.
- **Settings** (cog): font size, maximum levels and zebra rows — see Settings.
- **Connection indicator**: `LIVE`, `RECONNECTING` or `OFFLINE`, followed by the `host:port` of the `pm-api-gwy` instance the bridge reads — see Connection states.

18.5.2 Choosing a symbol

The picker works as `pm-viewer`'s does. Typing narrows the list **by prefix** (`A` offers the symbols that start with A, not every symbol containing one), ↑/↓ move the selection, **Enter** switches and **Esc** closes. A click works too.

Every book has its own URL: `/book/<SYMBOL>`, for example `http://localhost:8094/book/AAPL`. Bookmark it, or open several tabs on different symbols — each tab watches one symbol. The bare address `/` opens the book you viewed last in this browser, or the first listed symbol. A URL naming a symbol that is not listed shows `<SYMBOL> is not a listed symbol` instead of an empty book.

18.5.3 The statistics header

MSFT up 1.62%. This exchange has no previous close on record yet, so `Prev` is `n/a` and change is measured from the session open (`Basis session-open`). The green line under the header: the close is above that reference.

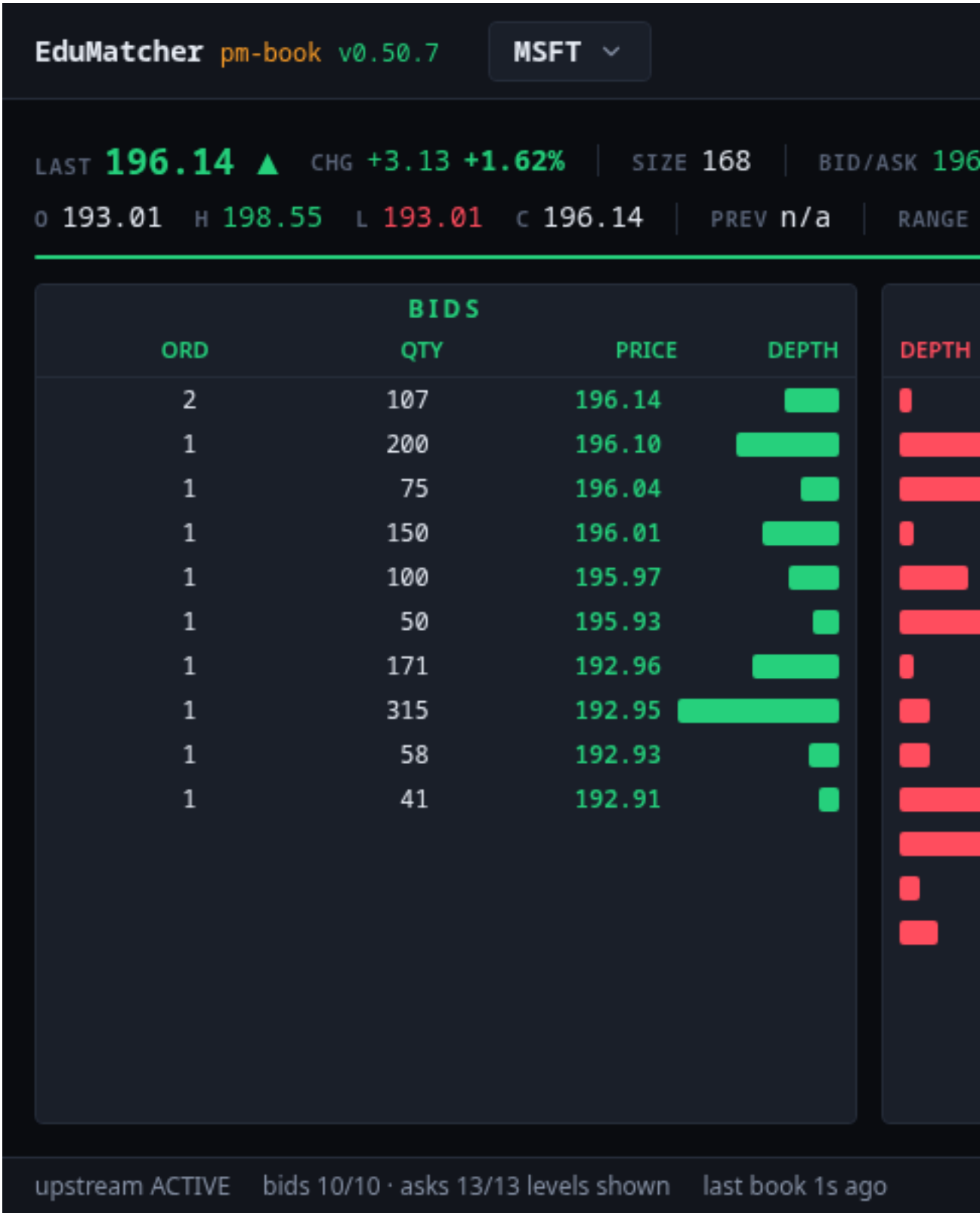


Figure 18.1: The order book viewer on MSFT, mid-session.



Figure 18.2: The statistics header.

Two rows, the same fields as `pm-viewer`'s header. The **reference** price that change and colour are measured against is the previous close when one is known, otherwise the session open.

Field	Meaning
Last	Price of the last trade, with / / against the reference, green when up and red when down
Chg	Last minus the reference, and the same as a percentage
Size	Quantity of the last trade
Bid/Ask	Best bid (green) × best ask (red)
Sprd	Ask minus bid
clock	Current time in the exchange session timezone (hover for the zone name)
O / H / L / C	The session's open, high, low and latest close, from every trade of the day
Prev	The previous trading day's close, `n/a` when there is none
Range	High minus low
Vol	Total quantity traded this session
Basis	What the change is measured against: `prev-close`, `session-open`, or `live since HH:MM`
date	Today's date in the exchange session timezone

A thin line under the header shows the session's direction: green when the close is above the reference, red when below, grey when level.

What live since HH:MM means

When the bridge cannot read the session's history — `pm-stats` is not running, or its database does not exist yet — it still shows the book and records every trade it sees from then on. The statistics are then correct only for that period, and the Basis field says so in amber, with the time the recording started. This is the web equivalent of running `pm-viewer` without a stats database. Times are then in UTC, because the session timezone also comes from `pm-stats`.

All times, the clock and the date use the exchange session timezone recorded by `pm-stats`, not the timezone of the browser — so a viewer in another country reads the same times as the exchange.

18.5.4 BIDS and ASKS

The two ladders mirror each other about the centre, as in `pm-viewer`, with the best price at the top of each:

BIDS columns	ASKS columns
Ord — number of resting orders at the level	**Depth** — bar, growing outward to the right
Qty — total quantity at the level	**Price**
Price	**Qty**
Depth — bar, growing outward to the left	**Ord**

The depth bars are scaled to the largest quantity among the levels **on screen**, so the largest visible level fills its bar. Prices use each symbol's own number of decimals from the reference data.

The ladders show the levels the gateway publishes (aggregated per price — never individual orders).

18.5.5 TRADES

The trade tape, newest first: **Time** (session timezone, to the millisecond), **Price** and **Qty**. Each print is coloured by its tick — green when its price is above the print before it, red when below, grey when equal.

When you open a book the tape already holds the session's earlier trades, read from history. The bridge keeps the newest `TAPE_MAX` (500) prints per symbol; the statistics count every trade of the session regardless.

18.5.6 How many levels are shown

The three panels share one row count. By default (**Max levels: Fit**) it is whatever fits the window, like `pm-viewer` in a terminal — make the window taller to see deeper into the book. The Settings popover can cap it at 10, 20 or 50 levels, like `pm-viewer --depth`.

The panels never reflow into a single column: in a narrow window the page scrolls sideways instead.

18.5.7 Footer

- **`upstream ACTIVE`** — the state of the bridge's connection to `pm-api-gwy`; shown in amber whenever it is anything else.
- **`bids 12/12 · asks 9/15 levels shown`** — how many of each side's levels are on screen, out of how many there are. A truncated ladder is never silent: `9/15` means six more ask levels exist below the fold.

- **`last book 2s ago`** — the age of the most recent book update. It keeps counting in a quiet market; that is normal, and the book shown is still current. `no book yet` means the symbol has not had a book since it was opened.
- **`s / F1 change symbol`** — the keyboard hint.

18.5.8 Settings

The cog in the top bar opens the settings. All of them are stored per browser.

Setting	Choices	Default	Notes
Font size	XS, S, M, L, XL, XXL	XS	Scales the whole page. Works in Chrome and Safari only
Max levels	Fit, 10, 20, 50	Fit	`pm-viewer`'s `--depth`
Zebra rows	on / off	off	`pm-viewer`'s `--zebra-lines`
Theme (top-bar button)	dark / light	dark	Light is meant for bright rooms and projectors

18.6 Connection states

Two connections are involved, and the viewer reports both:

Indicator	Meaning	What you see
`LIVE` (green)	The browser is connected to the bridge, and the bridge to `pm-api-gwy`	The live book
`RECONNECTING` (amber)	The browser is connected to the bridge, but the bridge's upstream connection is down or not yet authenticated	The last values received, with `upstream RECONNECTING` or `upstream DOWN` in the footer
`OFFLINE` (red)	The browser has lost its connection to the bridge	A *Disconnected from pm-book-bridge* banner; values are **hidden** rather than shown stale. The page reconnects by itself

The bridge reconnects upstream by itself, backing off from half a second to 15 seconds between attempts. When a working connection drops, the footer shows `upstream RECONNECTING` for the first 30 seconds (`UPSTREAM_DOWN`) and `upstream DOWN` after that. Before the first successful connection, and at once when `pm-api-gwy` rejects the key, it shows `upstream DOWN`.

No trade is lost across a short outage. On reconnect the bridge asks `pm-api-gwy` to resume each symbol's trades from the last one it saw. If the gap is longer than the gateway retains (`market_data_cache_sec`, 60 seconds by default — see API Gateway), the bridge rebuilds that symbol's statistics and tape from history instead, and logs that it did.

18.7 Configuration reference

The bridge is configured entirely with environment variables. Most installations set only `PM_BOOK_API_KEY` and, outside the whole stack, `API_GATEWAY_URL` and `LOG_SRV_HOST`.

18.7.1 Compose and host side

Variable	Default	Purpose
`BOOK_GUI_PORT`	`8094`	Host port the container is published on. Set it in `.env` for the whole stack, or on the command line for the app's own `make up`
`BIND_ADDR`	`127.0.0.1`	Whole stack only: host address the port is published on; `0.0.0.0` exposes it to the network
`EDUMATCHER_DATA_DIR`	—	App's own `make up` only: where to find `ref_data/engine_config.json` to read the key from

18.7.2 Bridge

Variable	Default (dev / container)	Purpose
`HOST` / `PORT`	`127.0.0.1` / `5194` — container `0.0.0.0` / `8094`	Bridge bind address and port
`CORS_ORIGIN`	*	CORS allow-list; restrict it to your site's origin when the viewer is exposed beyond localhost
`STATIC_DIR`	— (container: the built frontend)	Serve the built web UI from this directory
`MAX_WS_CLIENTS`	`200`	Maximum browser tabs; further connections are refused
`WS_HEARTBEAT_SEC`	`5`	How often every tab receives the bridge's status
`WS_PING_SEC` / `WS_PING_MAX_MISSED`	`10` / `2`	Ping to each tab, and missed replies before a dead tab is dropped
`WS_MAX_BUFFERED_BYTES`	`5000000`	A tab whose outgoing buffer exceeds this (a stalled browser) is disconnected
`TAPE_MAX`	`500`	Trades kept per watched symbol for the TRADES panel

18.7.3 Upstream (pm-api-gwy)

Variable	Default (dev / container)	Purpose
API_GATEWAY_URL	http://127.0.0.1:8081 — app compose http://host.docker.internal:8081, whole stack http://edumatcher:8081	Base URL of the dashboards instance. The market-data WebSocket URL is derived from it (ws://…/api/v1/ws/market-data, or wss:// for https://)
PM_BOOK_API_KEY	— (**required**)	The read-only (gateway_id: null) key of that instance. Never sent to the browser
UPSTREAM_PING_SEC / UPSTREAM_PING_MAX_MISSED	10 / 3	Liveness check of the upstream WebSocket
UPSTREAM_DOWN_AFTER_SEC	30	How long RECONNECTING lasts before it is reported as DOWN

18.7.4 Session phase (pm-md-gwy)

Variable	Default (dev / container)	Purpose
CALF_HOST / CALF_PORT	127.0.0.1 / 5570 — app compose host.docker.internal, whole stack edumatcher	pm-md-gwy, for the session phase in the top bar. No key needed
CALF_CLIENT_ID	pm-book-bridge	Name the bridge gives in CALF HELLO

18.7.5 Logging

Variable	Default (dev / container)	Purpose
LOG_SRV_ENABLED	true	false logs to stdout and skips even the startup probe
LOG_SRV_HOST / LOG_SRV_PORT	127.0.0.1 / 5600 — app compose host.docker.internal, whole stack edumatcher	pm-log-srv
LOG_SRV_CLIENT_ID	pm-book-bridge	Name the bridge registers with at the log server
LOG_SRV_INSTANCE	—	Optional suffix that tells several viewers' logs apart
LOG_CONNECT_TIMEOUT_SEC	0.5	Startup probe and each reconnect attempt

Variable	Default (dev / container)	Purpose
LOG_FAILOVER_TIMEOUT_SEC	30	How long a lost log-server connection may last before the bridge switches, for good, to a local file
LOG_QUEUE_MAXSIZE	2000	Log records buffered while reconnecting
LOG_FAILOVER_DIR	<data dir>/logs — container /app/logs	Where that local file goes

If pm-log-srv is not reachable at startup the bridge logs to stdout, which make logs or podman logs show.

18.8 Checking the bridge

GET /api/bridge/status on the viewer's port describes the bridge's state as JSON:

```
curl -s localhost:8094/api/bridge/status
```

Field	Meaning
upstream / since	ACTIVE, RECONNECTING or DOWN, and since when
source	The host:port of pm-api-gwy
symbols	How many symbols the reference data lists
watched	Each watched symbol, with the number of tabs watching it
wsClients	Connected browser tabs
logging	Where the bridge's logs are going right now

18.9 Troubleshooting

Symptom	Likely cause	What to check
make up stops with *EDUMATCHER_DATA_DIR is not set* or *No read-only credential*	There is no key to start with	Set EDUMATCHER_DATA_DIR, or pass PM_BOOK_API_KEY=...
Top bar stays RECONNECTING, footer upstream DOWN; the log says PM_BOOK_API_KEY is unset	The container was started without a key	Start it through make up, make up-all or ./edumatcher.sh start, or pass the key

Symptom	Likely cause	What to check
Top bar stays `RECONNECTING`; the log says `refused PM_BOOK_API_KEY`	The key is wrong for that instance — typically `API_GATEWAY_URL` points at `desk` (8080), or the key belongs to a different engine configuration	Point `API_GATEWAY_URL` at port 8081; after switching configuration, restart the whole stack so the key is read again
Top bar stays `RECONNECTING`, nothing in the log about the key	The bridge cannot reach `pm-api-gwy`	Check `API_GATEWAY_URL`; on Podman use `host.containers.internal`; check that `pm-api-gwy` is running
The session badge in the top bar stays `Unknown`	The bridge cannot reach `pm-md-gwy`, or it is not running	Check `CALF_HOST`/`CALF_PORT` and that TCP `5570` is reachable from the bridge; a badge that resolves after a second or two at exchange start is normal
Red *Disconnected from pm-book-bridge* banner	The browser cannot reach the bridge	Is the container running (`make ps`, `podman ps`)? Is the port right?
Basis says `live since HH:MM`; times are in UTC	History is unavailable — `pm-stats` is not running or has not created its database	Start `pm-stats`, then reload the page: a symbol's history is read again when it is next opened with no other tab watching it
Waiting for the symbol list… never goes away	The bridge could not read `/api/v1/reference/symbols`	See the rows about `RECONNECTING` above
`<SYMBOL> is not a listed symbol`	The URL names a symbol the exchange does not list	Press `s` and choose one
A ladder ends before the book does	More levels exist than fit the window, or **Max levels** caps them	Read the footer (`asks 9/15`); make the window taller or set Max levels to Fit
A container started with `make up` stays `RECONNECTING`; its log shows `127.0.0.1:8081` refusing connections	`API_GATEWAY_URL=http://127.0.0.1:8081` was left in the shell by `eval "$(make dev-env …)"`	`unset API_GATEWAY_URL PM_BOOK_API_KEY` and run `make up` again — see Local development
An older curl install has no viewer on 8094	`update` does not fetch a new `compose.yaml`	Re-run the one-line installer

18.10 Related documentation

- `pm-viewer` — Order Book Viewer — the terminal application this one mirrors
- API Gateway — the market-data WebSocket and history endpoints the bridge reads
- Statistics and Reporting — `pm-stats`, which records the history
- Installation — the container stack, `.env` and the read-only API key
- Trader Information Terminal — the sibling read-only display, with which it shares its structure and deployment conventions

- Centralized Log Server — the bridge's optional logging destination
- `docs-design/EduMatcher-Book-GUI.md` — the design document (repository checkout only)
- `web-apps/book-gui/README.md` — the implementation's own reference (repository checkout only)

CHAPTER 19

FAQ: Orders, Sessions and Positions

Questions that come up once you trade beyond the first day: why an order type was rejected or did not trigger, how average cost and saved orders behave, and what self-match prevention does. The first-day questions are in the Quick Start Guide's FAQ & Common Mistakes; how the exchange is installed, configured and operated is in the Operator's Guide's FAQ: Setup and Operation.

19.1 Order types

19.1.1 Why was my IOC order rejected?

IOC (Immediate-Or-Cancel) orders **cannot rest on the book**, so they are rejected during any non-matching phase (PRE_OPEN, OPENING_AUCTION, CLOSING_AUCTION, CLOSED). IOC is only valid during `CONTINUOUS` trading.

```
[09:20:11.403] REJECTED  6f5471b08c2d3eb970dc8c864bf270cb code=SESSION_NOT_PERMITTED  IOC orders not accepted dur
```

The same applies to `MARKET` and `FOK` orders.

Fix: Type `SESSION` in the trader console to see the current phase, then wait for `CONTINUOUS`, or use a `LIMIT` order with `TIF=DAY` if you are willing to have the order rest.

19.1.2 Why didn't my STOP order trigger?

STOP orders only trigger from **last trade price** events. If no trades have occurred since you submitted the STOP, the trigger is never evaluated — regardless of the current bid/ask quotes.

```
# This SELL STOP at 148 won't fire if no trades have happened yet
NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP|QTY=100|STOP=148.00
```

Fix: Check whether any trades have printed in the engine output (`--verbose` mode) or in the audit log. A stop only becomes relevant once the market is actively trading through its trigger level.

Also check the trigger direction. A **SELL STOP** triggers when `last_trade_price <= stop_price`. If the market is above your stop, it won't fire until price falls *to* or *through* it.

See Stop Trigger Logic for full details.

19.1.3 Why did my STOP_LIMIT order trigger but not fill?

This is expected behavior. When a STOP_LIMIT triggers, it converts to a **LIMIT order at your specified `PRICE=`**. If the market has moved past that limit price before the converted order can match, it sits on the book unfilled.

Example: SELL STOP_LIMIT with `STOP=148.00` and `PRICE=147.50`. If the market gaps down to $146.00 before the converted limit can fill, your order rests at $147.50 — but nobody is bidding that high. You are protected from filling at $146.00, but you may not fill at all.

Tradeoff: STOP gives fill certainty. STOP_LIMIT gives price certainty. Neither gives both.

19.1.4 My FOK order was rejected even though there was liquidity. Why?

FOK checks whether the **entire quantity** can be filled at the limit price *before* executing. It does not execute at all unless the full size is available.

```
# Book has only 80 shares at 150.00, but you want 100
NEW|SYM=AAPL|SIDE=BUY|TYPE=FOK|QTY=100|PRICE=150.00
# → REJECTED because 80 < 100
```

Fix: Reduce `QTY` to match available liquidity, or use a `LIMIT` order and accept a partial fill.

19.2 Positions and P&L

19.2.1 Why does my average cost change when I buy more shares?

When you add to an existing position, your **VWAP average cost** updates to reflect the new shares at the new price. This is standard practice for tracking the true cost basis of a position built over multiple trades.

Example:

```
Buy 100 shares @ 150.00  →  avg_cost = 150.00
Buy 100 shares @ 152.00  →  avg_cost = (150 × 100 + 152 × 100) / 200 = 151.00
```

See P&L & Clearing — VWAP Average Cost.

19.3 Saved orders

19.3.1 My GTC orders didn't reload after restart. Why?

GTC orders are written to `data/gtc_orders.json` **on engine shutdown** (when you press Ctrl-C or the scheduler sends a CLOSED transition). If the engine was killed forcefully (`kill -9` or a crash), the file was not written.

Fix: Check whether `data/gtc_orders.json` exists and has recent content. If the engine was killed mid-session, the file may be stale or missing.

19.4 Self-match prevention

19.4.1 What is SMP and why did my order get cancelled?

Self Match Prevention (SMP) stops you from accidentally trading against your own resting orders. If you submit a buy that would match a sell you already have on the same book, SMP kicks in.

```
# Submit a buy that would cross with your own resting sell
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=151.00|SMP=CANCEL_AGGRESSOR
```

With `SMP=CANCEL_AGGRESSOR`, the incoming buy is cancelled instead of filling against your own sell. Options:

Value	Behaviour
`NONE`	No SMP — you can trade against yourself
`CANCEL_AGGRESSOR`	The incoming order is cancelled
`CANCEL_RESTING`	The resting order is cancelled
`CANCEL_BOTH`	Both orders are cancelled

19.5 Prices

19.5.1 Why does the engine use integer ticks instead of float prices?

Float arithmetic can introduce tiny representation errors in comparisons and aggregation. Using integer ticks makes price-time matching and level math exact.