

Trading on Hyperliquid

Strategies, Market Making, and APIs for On-Chain Perps

Adrien Vosk

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Published by NobleTrex Press



© 2024 by N O B T R E X LLC. All rights reserved.
For permissions and other inquiries, write to:
P.O. Box 3132, Framingham, MA 01701, USA

This book is for educational and informational purposes only. The author and publisher make no guarantees about the accuracy or completeness of its contents and accept no liability for any loss or damage resulting from its use. Software tools such as large language model have been applied to assist in the writing and editing of this book. All product names, logos, and trademarks are the property of their respective owners. References to third-party products or names are for identification only and do not imply endorsement.

Contents

1	Foundations of On-Chain Perpetual Futures	5
1.1	Perpetual Futures Contract Basics	5
1.1.1	Perps vs. traditional futures: what changes when there is no expiry?	6
1.1.2	Perps vs. traditional futures: what changes when there is no expiry?	6
1.1.3	The three prices you must distinguish: index, mark, and last	7
1.1.4	The funding rate: the anchoring mechanism . .	8
1.1.5	Funding is not “free yield”: the basis-risk trade .	9
1.1.6	A minimal accounting model for perp economics	9
1.1.7	Funding cycles and convergence intuition	10
1.1.8	What a perp contract gives you operationally . .	11
1.1.9	Checklist: the perp basics your bot must get right	12
1.2	Orderbook Market Microstructure	13
1.2.1	The CLOB: two stacks of conditional liquidity .	13
1.2.2	Depth and slippage: why size changes your price	14
1.2.3	Price–time priority and queue position	15
1.2.4	Spread decomposition: why market making is not “set and forget”	15

1.2.5	Price impact and the Kyle lens	16
1.2.6	Microstructure metrics you will reuse in systematic market making	17
1.2.7	Why microstructure is the foundation for automated liquidity provision	17
1.3	Position, PnL, and Leverage Mathematics	18
1.3.1	State variables: the minimal position record . .	18
1.3.2	Notional exposure and leverage	19
1.3.3	Mark-to-market: price PnL for a single position	20
1.3.4	Funding accrual: PnL when the price does not move	21
1.3.5	Total PnL decomposition (what you should log)	21
1.3.6	Margin, initial vs maintenance, and why liquidation is mark-based	22
1.3.7	Liquidation intuition: leverage turns small moves into discontinuous outcomes	23
1.3.8	Cross margin vs isolated margin: why liquidation thresholds can drift	23
1.3.9	Continuous monitoring: what an automated system should observe	24
1.3.10	Implementation checklist (math you should unit-test)	24
1.4	Core Risk Metrics for Traders and Quants	25
1.4.1	Returns, variance, and volatility: the baseline units of risk	26
1.4.2	Maximum drawdown: a path-dependent risk that traders actually feel	27
1.4.3	Tail events and gap risk: why stop-loss logic can fail	28
1.4.4	Distance to liquidation as a risk budget	28

1.4.5	VaR and Expected Shortfall (CVaR): measuring tails correctly	29
1.4.6	Stress tests: the risk tool that does not lie	30
1.4.7	Operational safeguards for automated strategies	30
1.4.8	A practical risk dashboard: what to compute every tick	32
1.5	Blockchain Execution Principles	32
1.5.1	Deterministic state transitions vs. uncertain ordering	33
1.5.2	Finality and reorg profile: why HyperBFT matters	33
1.5.3	Latency decomposition: where time is spent . .	34
1.5.4	Transaction ordering and MEV: adversarial execution is part of the game	35
1.5.5	Fee markets and inclusion probability	35
1.5.6	A chain-aware execution cost model	36
1.5.7	Engineering patterns: making execution robust	37
1.5.8	Summary: what on-chain execution changes for traders	38
2	Hyperliquid Architecture and Market Mechanics	39
2.1	Hyperliquid L1 Architecture and HyperEVM	39
2.1.1	Two execution environments, one venue state .	40
2.1.2	What “not a separate chain” means in practice .	41
2.1.3	Why traders should care: latency, determinism, and failure modes	41
2.1.4	Composability: what you can assume without over-assuming	42
2.1.5	Design takeaway for bots and on-chain strategies	43
2.2	HyperCore Components and Consensus	43

2.2.1	From order submission to final state: what gets decided where	44
2.2.2	Consensus: why finality changes how you engineer execution	44
2.2.3	Execution primitive: deterministic fills from a central limit book	45
2.2.4	Clearinghouse: turning fills into portfolio constraints	45
2.2.5	Pricing inputs: oracle, mark construction, and why venue liquidity matters	46
2.2.6	Finality versus probabilistic confirmation: trading implications	47
2.2.7	Closed-loop intuition: price feeds, venue state, and risk actions	48
2.3	Instrument Specifications and Identifiers	48
2.3.1	Contract specification: what a “perp contract” is on Hyperliquid	49
2.3.2	Identifiers: symbols are not enough	49
2.3.3	A bot-grade resolution workflow (metadata first, orders second)	50
2.3.4	Precision constraints: sizes are decimal-quantized, prices are precision-limited	51
2.3.5	Why these rules exist: discrete books and deterministic risk	52
2.3.6	Operational failure modes and how to eliminate them	53
2.3.7	Minimal checklist for “order validity” in automated execution	53
2.4	Margining and the Perpetual Risk Engine	54
2.4.1	Cross margin vs isolated margin: coupling versus containment	55

2.4.2	Initial margin: the admission ticket for new exposure	55
2.4.3	Maintenance margin: the survival threshold . .	56
2.4.4	Equity decay, gap risk, and liquidation triggers .	57
2.4.5	A simple equity-versus-threshold picture	57
2.4.6	Backstop liquidation and why cross vs isolated outcomes differ	58
2.4.7	Modeling guidance: what to include in sizing and execution logic	58
2.5	Funding Mechanics and Strategy Implications	59
2.5.1	Peer-to-peer transfers on an hourly clock	60
2.5.2	Where the rate comes from: interest + premium, premium from impact vs oracle	60
2.5.3	Notional convention: funding uses oracle price, not the venue's mark	61
2.5.4	Funding caps: bounding extreme carry	62
2.5.5	PnL decomposition: separate price PnL from funding cashflows	62
2.5.6	Strategy implications	63
2.5.7	Implementation note for bots: model funding as a scheduled cashflow	64
2.6	Fee Structures and Rebates	65
2.6.1	The fee state is account-specific and time-varying	65
2.6.2	Maker rebates and taker fees: the sign matters .	66
2.6.3	HIP-3 markets: fee economics may differ by market family	67
2.6.4	Turnover math: expected value with fee and rebate terms	67
2.6.5	Why execution constraints feed back into fee economics	68

2.6.6	A simple break-even framework for high-turnover liquidity provision	68
2.6.7	Implementation checklist: fee-aware trading and backtesting	69
3	Order Execution and Microstructure Tactics	71
3.1	Matching Priority and Fill Dynamics	71
3.1.1	Price–time priority as a queueing problem . . .	72
3.1.2	Partial fills and the “fill path”	73
3.1.3	Maker vs taker fills: knowing what you actually did	74
3.1.4	A practical fill-probability model (and what it leaves out)	75
3.1.5	Adverse selection: why passive fills can be costly even with rebates	76
3.1.6	Queue-position tactics that do not accidentally increase toxicity	77
3.1.7	Join vs improve: when one tick matters	77
3.1.8	Cancel/replace without “thrashing” your queue	78
3.1.9	Do not “optimize queue” by quoting through unstable microstructure	79
3.1.10	Quantifying your queue and toxicity from streams	79
3.1.11	Microstructure “gotchas” that show up only in production	80
3.1.12	A compact playbook	80
3.2	Core Order Types	81
3.2.1	A precise vocabulary: instruction, evaluation, and outcome	81
3.2.2	Market orders: deterministic immediacy, non-deterministic price	82
3.2.3	Limit orders: price control, queue uncertainty .	83

3.2.4	Trigger orders: Stop vs Take, Market vs Limit . . .	85
3.2.5	Stop versus Take: what is the condition?	85
3.2.6	Market versus Limit after trigger: certainty versus control	85
3.2.7	Triggers as part of a position lifecycle	86
3.2.8	Order taxonomy: intent-to-mechanics map . . .	87
3.2.9	API-level semantics: the minimal fields you must reason about	87
3.2.10	Choosing the right primitive: three decision questions	88
3.3	Protective Order Flags	89
3.3.1	Post-only (ALO): maker-only as an enforceable constraint	89
3.3.2	Reduce-only: exposure cannot increase, even if you mis-size	91
3.3.3	Combining ALO and reduce-only: constrained execution intent	92
3.3.4	A logic tree for how the venue enforces these constraints	92
3.3.5	Common edge cases and how to defend against them	93
3.3.6	Stale position state when sending exits	93
3.3.7	Post-only rejections due to microprice drift . . .	93
3.3.8	Trigger orders that behave like markets during stress	93
3.3.9	Monitoring: confirming your protections worked	94
3.3.10	Summary: constraints as part of strategy design	94
3.4	Advanced Execution Orders	95
3.4.1	Why slicing works: impact is convex in size . . .	95

3.4.2	Venue-native TWAP: deterministic cadence and constraints	96
3.4.3	TWAP failure modes you must plan for	97
3.4.4	Transparency and sunshine trading risk	98
3.4.5	Connecting TWAP knobs to the Almgren-Chriss view	98
3.4.6	Order-flow-aware schedules: when constant rate is not optimal	99
3.4.7	Scale orders: distributing a limit schedule across prices	100
3.4.8	Native versus custom: a decision checklist	101
3.4.9	Monitoring TWAP and Scale progress	101
3.4.10	Advanced orders as controlled exposure to the book	102
3.5	Execution Tactics	102
3.5.1	The two-mode view: passive capture vs aggressive completion	103
3.5.2	Microstructure heuristics that actually map to decisions	103
3.5.3	Spread and “paying up” threshold	103
3.5.4	Volatility/markout filter for maker exposure	104
3.5.5	Order-flow imbalance (toxicity proxy)	104
3.5.6	Deadline-driven escalation: passive first, then controlled aggression	105
3.5.7	Order management: robust loops without self-inflicted queue loss	106
3.5.8	Avoid unnecessary churn: when to update a quote	106
3.5.9	Controlled aggression: IOC rather than full market when possible	106
3.5.10	State management: reconcile from fills, not from intent	107

3.5.11	Use the fill stream as the execution source of truth	107
3.5.12	Handle partial fills as a first-class state	107
3.5.13	Sync and resync: surviving missed messages . .	107
3.5.14	Putting it together: a minimal execution policy template	108
3.5.15	Capstone: a production-style execution state machine	109
3.5.16	Implementation notes: invariants to enforce in code	109
4	Market Data, Simulation, and Quant Workflows	111
4.1	Market Data Types for Orderbook Strategies	111
4.1.1	The core data modalities (minimum viable microstructure dataset)	112
4.1.2	L2 order book: snapshots + incremental deltas .	112
4.1.3	Trade prints (public): the only ground truth for aggressor flow	113
4.1.4	Funding series (perps-specific carry and PnL driver)	114
4.1.5	Metadata and context: metaAndAssetCtxs as “market configuration”	114
4.1.6	Public vs. private data: what you need for research vs. what you need for reconciliation . . .	115
4.1.7	Timestamping, ordering, and determinism . . .	115
4.1.8	Use multiple clocks and keep them separate . .	115
4.1.9	Assign your own sequence numbers	116
4.1.10	Dropped-message monitoring is not optional . .	116
4.1.11	Storage design: event logs first, derived bars second	117
4.1.12	Partitioning and indexing for HFT backtests . .	117
4.1.13	Data retention and compression	117

4.1.14	A robust ingestion pipeline (from WebSocket to research-grade databases)	118
4.1.15	Operational checklists for “research-grade” capture	119
4.1.16	Completeness checks	120
4.1.17	Correctness checks	120
4.1.18	Performance checks	120
4.1.19	A compact schema blueprint (what to store and why)	121
4.1.20	The guiding principle: archive what you would have known at the time	121
4.2	Backtesting and Fill Simulation	122
4.2.1	What “realistic” means for an HFT backtest . . .	122
4.2.2	A fill-model ladder: from toy assumptions to research-grade simulators	123
4.2.3	Latency modeling: turning market data into “as-observed” state	124
4.2.4	Three clocks and two latencies	124
4.2.5	Why latency creates non-deterministic fills . . .	124
4.2.6	Queue and priority approximations without a full matching engine	125
4.2.7	State variables that matter	125
4.2.8	A robust reduced-form model for passive fills . .	126
4.2.9	Order lifecycle in simulation: placement, partial fills, and cancellation races	126
4.2.10	Sequence of events you must represent	126
4.2.11	Sequence diagram: simulated order lifecycle with latency and probabilistic fills	127
4.2.12	Partial fills and execution price formation	127

4.2.13	Non-determinism: embrace it, but make it reproducible	128
4.2.14	Calibration and drift: linking fill intensities to market state	128
4.2.15	Unit tests for a book builder and fill simulator	129
4.2.16	Orderbook reconstruction tests	129
4.2.17	Execution-simulator tests	130
4.2.18	A practical architecture: event-driven replay + modular execution models	130
4.2.19	When to stop adding realism	130
4.3	The Strategy Research and Iteration Loop	131
4.3.1	The research loop as an engineering system	131
4.3.2	Why this loop matters specifically on on-chain perps	132
4.3.3	Stage 1: Hypothesis and falsifiable predictions	132
4.3.4	Stage 2: Dataset build (streaming vs batch/backfill)	133
4.3.5	Streaming capture (for correctness)	133
4.3.6	Batch/backfill (for coverage and long windows)	133
4.3.7	Schema constraints are part of the dataset	134
4.3.8	Stage 3: Backtest design that resists overfitting	134
4.3.9	Use strict evaluation windows (walk-forward)	134
4.3.10	Keep the fill model fixed while testing signal changes	135
4.3.11	Prefer small, interpretable parameter sets	135
4.3.12	Stage 4: Attribution review as the gatekeeper	136
4.3.13	Stage 5: Experiment tracking (make iteration cumulative)	136
4.3.14	Stage 6: Paper trading as a simulator audit	137
4.3.15	Reality-gap diagnostics	137

4.3.16	A/B changes with incremental rollout	137
4.3.17	Stage 7: Production promotion criteria and monitoring for decay	138
4.3.18	Promotion checklist	138
4.3.19	Monitoring for structural change	138
4.3.20	Putting it together: a workflow that scales . . .	139
4.4	Regime Detection and Volatility Filters	139
4.4.1	What a “regime” means for an orderbook strategy	140
4.4.2	Feature design: a small state vector that is hard to fool	140
4.4.3	Volatility: candle-based realized volatility	141
4.4.4	Liquidity and microstructure: spread, depth, and imbalance	141
4.4.5	Flow intensity: trade arrival rate spikes	141
4.4.6	Context features from metaAndAssetCtxs: impact, funding, premium, open interest	142
4.4.7	Two layers of filters: risk scaling and hard stops	142
4.4.8	Volatility scaling	143
4.4.9	Liquidity scaling using spread and impact proxies	143
4.4.10	Hard stops (algorithmic on/off switches)	143
4.4.11	Online changepoint detection (BOCPD) as a regime engine	144
4.4.12	What BOCPD provides	144
4.4.13	What to feed into BOCPD	144
4.4.14	Using the run-length distribution for controls .	145
4.4.15	Connecting regimes to quoting parameters (a control map)	145
4.4.16	Avoiding a common trap: regime filters that overfit	146
4.4.17	Guardrail 1: Evaluate filters by stability, not only PnL	146

4.4.18	Guardrail 2: Keep regimes coarse	146
4.4.19	Guardrail 3: Validate with reality-gap diagnostics	146
4.4.20	A practical implementation order	147
4.5	Performance Attribution	147
4.5.1	Attribution: what it is and what it is not	148
4.5.2	The minimal attribution tree for on-chain perps	148
4.5.3	Compute the basic PnL components	149
4.5.4	1) Trading PnL from fills	149
4.5.5	2) Fees and maker rebates	150
4.5.6	3) Funding PnL	150
4.5.7	4) Inventory mark-to-market	150
4.5.8	Implementation shortfall (IS): the execution- centric benchmark	151
4.5.9	Decision price and executed price	151
4.5.10	Total implementation shortfall	151
4.5.11	Microstructure-aware IS for continuous quoting	152
4.5.12	Reconciling backtest attribution with live attri- bution	152
4.5.13	The reconciliation dataset	153
4.5.14	A practical discrepancy checklist	153
4.5.15	A cumulative PnL breakdown (example chart) .	154
4.5.16	Using attribution to improve the strategy (not just explain it)	154
5	Systematic Trading and Market Making Strategies	155
5.1	Directional Strategies	156
5.1.1	A funding-aware objective: edge must beat carry & friction	156
5.1.2	Practical funding expectation	157

5.1.3	From signal to trade: a funding-aware decision rule	157
5.1.4	A general decision inequality	158
5.1.5	A compact “funding-adjusted alpha”	158
5.1.6	Signal families translated: momentum vs mean reversion	158
5.1.7	Momentum: hold only when expected drift dominates funding drag	159
5.1.8	Funding-aware momentum holding window . .	159
5.1.9	Entry style	160
5.1.10	Mean reversion: funding can be your friend or your enemy	160
5.1.11	A simple asymmetric threshold	160
5.1.12	Position sizing under leverage: liquidation-aware, funding-aware	161
5.1.13	Sizing from a target risk budget	161
5.1.14	Funding-aware shrinkage	161
5.1.15	Hard caps: leverage & concentration limits . . .	162
5.1.16	Execution on Hyperliquid: making signal strength survive slippage	162
5.1.17	Urgency ladder: ALO vs IOC vs TWAP	162
5.1.18	Funding-aware scheduling: align rebalancing to funding boundaries	163
5.1.19	Concrete algorithms	163
5.1.20	Algorithm 1: Funding-filtered momentum with time-decaying exposure	164
5.1.21	Inputs	164
5.1.22	Core logic	164
5.1.23	Why this works	165

5.1.24	Algorithm 2: Asymmetric mean reversion that “prefers” the funded side	165
5.1.25	Inputs	165
5.1.26	Core logic	165
5.1.27	Why this works	166
5.1.28	Implementation checklist: making the strategy robust in production	166
5.1.29	Pseudocode: a funding-aware directional execution loop	167
5.1.30	Common failure modes (and how funding-awareness prevents them)	168
5.2	Funding and Carry Optimization Strategies	168
5.2.1	Delta-neutral framing: turn funding into a targeted yield stream	169
5.2.2	Basis drift risk: the hedge breaks through $P_t - S_t$	170
5.2.3	Designing the carry book: what you optimize	171
5.2.4	A concrete baseline strategy: long spot / short perp carry	172
5.2.5	Sizing: notional first, then quantity	172
5.2.6	Entry filter: fund-and-premium consistency	173
5.2.7	Re-hedging and action budgets: staying neutral without overtrading	173
5.2.8	A simple delta estimator	174
5.2.9	Basis risk controls: limits, stops, and “basis VaR”	174
5.2.10	Hard basis limit	174
5.2.11	Basis VaR approximation	175
5.2.12	Margin and liquidation constraints on the perp leg	175
5.2.13	A practical leverage policy for carry	175
5.2.14	Margin-aware re-hedge priority	176
5.2.15	Execution details: minimizing hedge slippage	176

5.2.16	Sequencing and “leg risk”	176
5.2.17	Order types as cost controls	176
5.2.18	Carry selection across assets: harvesting asymmetry without concentration	177
5.2.19	Operational blueprint: a carry bot that respects risk and action limits	177
5.2.20	Checklist: what to log and monitor	179
5.2.21	Takeaways	179
5.3	Basis and Arbitrage Execution	180
5.3.1	The object you trade: basis as a priced risk	181
5.3.2	From “mid basis” to basis-at-fill	181
5.3.3	Profit model: convergence plus carry, minus frictions	182
5.3.4	Execution sequencing risk: the arb lives in the time gap	182
5.3.5	Leg risk taxonomy	182
5.3.6	Cross-venue execution risk	183
5.3.7	Two guardrails for sequencing	183
5.3.8	Guardrail 1: timeouts and edge decay	183
5.3.9	Guardrail 2: dynamic hedge urgency	184
5.3.10	Practical arbitrage patterns	184
5.3.11	Pattern A: Cross-venue spot–perp convergence	184
5.3.12	Pattern B: Cross-venue perp–perp spread	185
5.3.13	Pattern C: Intravenue proxy basis using oracle and impact	185
5.3.14	Choosing order types under microstructure and rate limits	185
5.3.15	IOC vs post-only as a completion dial	186
5.3.16	Action-budget-aware batching	186

5.3.17	Rebalancing around hourly funding: discrete constraints, continuous signals	186
5.3.18	A robust arbitrage decision engine	187
5.3.19	Layer 1: Edge estimation in fill space	187
5.3.20	Layer 2: Sequencing penalty	187
5.3.21	Layer 3: Funding term (if holding)	188
5.3.22	Operational checklist: what breaks arbs in practice	188
5.3.23	Takeaways	188
5.4	Market Making Edge Decomposition	189
5.4.1	The core unit: a single maker fill	190
5.4.2	Immediate PnL vs subsequent PnL	190
5.4.3	From micro to macro: a PnL decomposition over time	191
5.4.4	Realized spread: what you actually capture . . .	191
5.4.5	Fees and maker rebates: a systematic term on Hyperliquid	192
5.4.6	Toxicity: measuring the hidden loss with markout	192
5.4.7	A compact EV model for continuous quoting . .	193
5.4.8	Adding funding: inventory has a carry term on perps	194
5.4.9	Estimating components in practice (without fragile assumptions)	195
5.4.10	Step 1: Label trades as maker vs taker and store microstate	195
5.4.11	Step 2: Compute markouts (toxicity) on a horizon grid	195
5.4.12	Step 3: Compute net spread in dollars	195
5.4.13	Step 4: Attribute funding on residual inventory	196
5.4.14	Regime intuition: when each term dominates .	196
5.4.15	Low volatility / stable order flow	196

5.4.16	High volatility / jumpy microstructure	196
5.4.17	A minimal “edge report” template	197
5.4.18	Takeaways	197
5.5	Inventory Control and Risk Models	198
5.5.1	A state variable: inventory as the control knob .	198
5.5.2	Reservation price: skew to make inventory mean-revert	199
5.5.3	A practical linear skew	199
5.5.4	Why “skew both sides” is better than moving only one side	200
5.5.5	Spread control: widen when inventory and un- certainty rise	200
5.5.6	Probabilistic view: controlling fill rates	201
5.5.7	Inventory bands: soft and hard limits	201
5.5.8	Forced de-risk: choose certainty over rebates . .	202
5.5.9	Funding-aware inventory control: inventory has carry	202
5.5.10	Funding-adjusted inventory target	202
5.5.11	Inventory skew explanation (with depth chart vi- sual)	203
5.5.12	A liquidation-aware risk overlay	203
5.5.13	Stress-based distance metric	204
5.5.14	Mapping overlay to actions	204
5.5.15	Putting it together: a complete inventory control policy	204
5.5.16	Implementation notes for automated bots . . .	205
5.5.17	Update thresholds	205
5.5.18	Separate maker mode from de-risk mode	205
5.5.19	Takeaways	206

5.6	Quoting Algorithms	206
5.6.1	Data inputs: WebSocket-first with snapshot reconciliation	207
5.6.2	Fair value from L2: microprice and imbalance	208
5.6.3	Microprice (L1 depth-weighted mid)	208
5.6.4	Extending to L2: multi-level imbalance	208
5.6.5	Practical calibration	209
5.6.6	Fair value from trades: signed order flow and short-term momentum	209
5.6.7	Signed trade imbalance	209
5.6.8	A blended estimator	210
5.6.9	Volatility estimation for quoting: microstructure-relevant, not daily	210
5.6.10	EWMA of mid returns	210
5.6.11	Jump sensitivity and vol floors	211
5.6.12	Quoting ladders: two-level recipes that trade off edge and risk	211
5.6.13	Two-level ladder	211
5.6.14	Size allocation across levels	212
5.6.15	Inventory-linked quote selection: fade-to-flat as a mode switch	212
5.6.16	Normal mode ($ Q_t \leq Q_{fade}$)	212
5.6.17	Fade-to-flat mode ($ Q_t > Q_{fade}$)	212
5.6.18	Order management on Hyperliquid: making quotes safe under cancels and limits	213
5.6.19	Thresholded quote updates	213
5.6.20	Post-only default with a safety valve	213
5.6.21	Batching and address-based action budgets	214
5.6.22	A reference quoting loop (pseudocode)	214

5.6.23	Practical calibration workflow	215
5.6.24	Takeaways	216
5.7	Protocol Liquidity and HLP Vaults	216
5.7.1	What HLP is (economically)	217
5.7.2	HLP as “liquidity beta”: mapping to the EV decomposition	218
5.7.3	Impact on liquidity depth and organic spread contraction	219
5.7.4	HLP and liquidation handling: why it matters for both traders and depositors	219
5.7.5	1) Liquidity continuity in stress	220
5.7.6	2) A stress-linked risk premium for HLP depositors	220
5.7.7	Lock-ups, correlation, and portfolio design . . .	220
5.7.8	HLP as a strategic sleeve	221
5.7.9	Drawdown clustering	221
5.7.10	HLP versus running your own maker strategy: a decision framework	221
5.7.11	When HLP is attractive	221
5.7.12	When proprietary market making is attractive .	222
5.7.13	A hybrid approach	222
5.7.14	How HLP shapes your execution costs (practical implications)	223
5.7.15	A concise checklist for depositors	223
5.7.16	Takeaways	223
6	API Integration and Exchange Connectivity	225
6.1	API Architecture and Environments	226
6.1.1	Two API surfaces: Info (read) versus Exchange (write)	227

6.1.2	REST versus WebSocket: what each transport is for	227
6.1.3	Environment selection: mainnet versus testnet .	228
6.1.4	A safe client mental model: the “read mirror” and the “write actuator”	229
6.1.5	Practical endpoints and transports to implement first	230
6.2	Cryptographic Authentication	231
6.2.1	What “ECDSA authentication” means in this context	231
6.2.2	Why SDK parity matters (and why signature bugs look random)	232
6.2.3	Threat model: what you are defending against .	232
6.2.4	Typed structured signing versus legacy signing .	233
6.2.5	Signing pipeline: isolate responsibilities	233
6.2.6	Sequence of a signed action	234
6.2.7	Key management: how to sign without exposing private keys	234
6.2.8	Signature failure taxonomy (what breaks most often)	235
6.2.9	Implementation discipline: a minimal signing interface	236
6.2.10	How this connects to the rest of the chapter . . .	236
6.3	Nonces and API Wallets	237
6.3.1	API wallets (agent wallets): delegation as a security primitive	237
6.3.2	What a nonce is, and why exchanges insist on it	238
6.3.3	Hyperliquid’s nonce rules (what is actually enforced)	239
6.3.4	Designing a nonce allocator that works under concurrency	240

6.3.5	Nonce collisions: how they happen in real systems	241
6.3.6	API wallet lifecycle risks: pruning, deregistration, and address reuse	242
6.3.7	Connecting nonces and agent wallets back to system architecture	243
6.4	WebSocket Synchronization	243
6.4.1	The core WS contract: subscriptions, snapshots, and streaming updates	244
6.4.2	Why an FSM beats “event handlers”	244
6.4.3	Local L2 book building: snapshot + deltas with “zero-copy” discipline	245
6.4.4	A synchronization FSM for market streams . . .	246
6.4.5	State meanings and hard rules.	246
6.4.6	Disconnects are normal: engineer for periodic drops	247
6.4.7	Heartbeats and liveness	247
6.4.8	WS post: request/response correlation without leaving the socket	248
6.4.9	Rate limits: design within WS constraints . . .	249
6.4.10	Putting it together: a state sync loop you can reason about	249
6.5	Order Lifecycle and Reconciliation	250
6.5.1	Order identifiers: exchange oid versus client cloid	250
6.5.2	Submission outcomes: what the Exchange API can tell you immediately	251
6.5.3	Time-in-force and why it changes lifecycle dynamics	251
6.5.4	A client-side order state model that reconciles cleanly	252
6.5.5	User streams: the fast path for reconciliation . .	253

6.5.6	Info queries: the authoritative backstop	254
6.5.7	A reconciliation playbook (battle-tested workflow)	254
6.5.8	Troubleshooting matrix: “why did my order disappear?”	256
6.5.9	Idempotency: what you can and cannot guarantee	256
6.5.10	Putting reconciliation into your execution engine	257
6.6	Language Selection and SDK Ecosystem	258
6.6.1	Start from requirements, not from language preference	258
6.6.2	Python: fastest path to correctness and iteration	259
6.6.3	TypeScript: strong fit for Node services and front-end integration	260
6.6.4	Rust: performance, memory safety, and concurrency control	261
6.6.5	Native SDKs versus CCXT: portability versus venue-specific control	261
6.6.6	Decision table: choosing a stack for different bot profiles	262
6.6.7	A minimal “hello world” checklist for any language	263
6.6.8	Closing perspective	263
7	Bot Architecture, Operations, and Production Security	265
7.1	Event-Driven Bot Architecture	265
7.1.1	The core principle: make time explicit with events	266
7.1.2	Define contracts: events, state, and intents . . .	267
7.1.3	A practical event taxonomy	268
7.1.4	The message bus: buffer, sequence, and apply backpressure	268
7.1.5	Ingestion: normalize early, decide late	269
7.1.6	Snapshot-first streams: hydrate, then increment	269

7.1.7	Normalize identifiers for idempotency (OID vs CLOUD)	270
7.1.8	State ownership: one ledger, many readers . . .	270
7.1.9	Persistence: checkpoints, not journals (unless you need replay)	270
7.1.10	Alpha engine: produce intents, never touch the wire	271
7.1.11	Risk engine: a gate with memory	272
7.1.12	Execution engine: asynchronous I/O with idempotent order management	273
7.1.13	Single point of retry	273
7.1.14	Reconciliation: repair truth after partial failure	274
7.1.15	A concrete skeleton: queues between components	274
7.1.16	Graceful recovery: startup, reconnect, and shutdown sequences	276
7.1.17	Startup (cold start)	276
7.1.18	Reconnect (hot recovery)	276
7.1.19	Shutdown (safe stop)	277
7.1.20	What decoupling buys you in practice	277
7.2	Rate Limits and Error Resilience	278
7.2.1	Start with a failure taxonomy: transient, terminal, and ambiguous	278
7.2.2	One retry layer, bounded budgets, and short timeouts	279
7.2.3	Timeouts: fail fast so you can recover	279
7.2.4	Retry budgets: cap both count and time	280
7.2.5	Backoff with jitter: avoid synchronized retry bursts	280
7.2.6	Determinism where it matters	281
7.2.7	Adaptive throttling using /info userRateLimit . .	281

7.2.8	The expiresAfter trap: avoid self-amplifying throttling	282
7.2.9	Circuit breakers: stop digging when the dependency is failing	283
7.2.10	What should trip the breaker?	283
7.2.11	What should the breaker do?	284
7.2.12	Bulkheads and time limiters: isolate damage within your own bot	284
7.2.13	Idempotency and reconciliation: the safe alternative to blind retries	285
7.2.14	A minimal, deterministic resilience policy (recipe)	286
7.2.15	What to log and measure (so resilience is testable)	286
7.2.16	Putting it together: safe behavior under stress .	287
7.3	System Latency Optimization	287
7.3.1	What to optimize: end-to-end latency, not a single stopwatch	288
7.3.2	Use distributions, not averages	288
7.3.3	Tail latency: why it dominates and where it comes from	289
7.3.4	Concurrency model choices: keep fast paths uncontended	289
7.3.5	Single-writer state: a simple rule that prevents locks	289
7.3.6	Event loop vs thread pool: choose based on blocking behavior	290
7.3.7	Backpressure is also a latency tool	290
7.3.8	Measurement infrastructure: make latency attributable	291
7.3.9	Network-path latency: control what you can, instrument the rest	292

7.3.10	TCP behavior: small messages and Nagle's algorithm	292
7.3.11	Connection reuse and warm paths	292
7.3.12	Host and runtime tuning: reduce jitter before chasing microseconds	293
7.3.13	Batching and throttling: latency versus control .	293
7.3.14	Latency and execution quality: optimize the right objective	294
7.3.15	A step-by-step optimization workflow	294
7.4	Testnet Validation and Simulation Workflows	295
7.4.1	The layered pipeline: unit tests, replay simulation, testnet acceptance, then mainnet	295
7.4.2	Deterministic unit tests: making the bot testable by design	296
7.4.3	Test the state machine, not the network	296
7.4.4	Control time explicitly in tests	297
7.4.5	Floating-point hygiene: test rounding at boundaries	297
7.4.6	Replay-based simulation: validating behavior under realistic sequences	297
7.4.7	Build an append-only event log	298
7.4.8	Replays should be deterministic even when the original run was not	298
7.4.9	Use replays to test stress behaviors	299
7.4.10	Testnet validation: confirm real exchange semantics without risking capital	299
7.4.11	Environment separation: treat testnet as a first-class target	299
7.4.12	Test funds: operationalize funding like any other dependency	300

7.4.13	Acceptance tests: the minimal end-to-end smoke suite	300
7.4.14	Chaos testing before mainnet: break your bot on purpose	301
7.4.15	CI gates: making correctness the default	301
7.4.16	Pre-deploy checklist: a short list that prevents most disasters	302
7.5	Health Monitoring and Alerts	302
7.5.1	Observability goals: detect symptoms early and explain them later	303
7.5.2	Telemetry architecture: metrics, logs, and traces with correlation	303
7.5.3	Standardize identifiers for correlation	304
7.5.4	What to monitor: a symptom-first checklist	304
7.5.5	Alert design: high signal, anti-flap, and runbooks	305
7.5.6	Tie every page to a runbook	305
7.5.7	Metamonitoring: monitor the monitoring	306
7.5.8	Automated kill switch: design goals and safety properties	306
7.5.9	Telemetry pipeline feeding the kill switch	307
7.5.10	Decision engine: from signals to actions	307
7.5.11	Actuation safety: do not create more failure load	308
7.5.12	Kill-switch schematic	308
7.5.13	Concrete kill-switch triggers (examples)	308
7.5.14	Testing the monitoring and kill switch	309
7.5.15	Operational discipline: keep the system understandable	309
7.6	Key Management and Production Security	310
7.6.1	Threat model first: defend against how keys are actually stolen	310

7.6.2	Principle of least privilege: reduce what any single key can do	311
7.6.3	Two-tier key model: trading hot key vs treasury cold funds	311
7.6.4	Privilege by environment: testnet keys must never touch mainnet	312
7.6.5	Secure secret storage and injection: do not ship keys with code	312
7.6.6	Architecture: encrypted vault injection into a restricted runtime	313
7.6.7	Signing isolation patterns: local hardened host vs delegated signing	313
7.6.8	Pattern A: local signing on a hardened host . . .	313
7.6.9	Pattern B: delegated signing via Vault Transit / KMS-backed service	314
7.6.10	Secrets handling in practice: the "never leak" checklist	314
7.6.11	Runtime protections: reduce the blast radius of compromise	315
7.6.12	Incident response playbook: assume compromise is possible	316
7.6.13	1) Containment: stop signing and stop trading .	316
7.6.14	2) Key rotation: replace compromised authority	316
7.6.15	3) Forensics: preserve evidence without leaking secrets	316
7.6.16	4) Restore: resume in constrained mode	317
7.6.17	Reasoning about wallet security systematically .	317
7.6.18	Summary: the security posture that survives real operations	318

Introduction

The architecture of modern financial markets is undergoing a profound and accelerating transformation. For years, the cryptocurrency trading landscape was defined by a stark dichotomy. On one side stood centralized exchanges (CEXs)—platforms offering deep liquidity, low latency, and advanced order types, but demanding that users surrender custody of their assets to opaque institutional entities. On the other side stood decentralized exchanges (DEXs)—venues that guaranteed cryptographic self-custody and deterministic transparency, but suffered from the sluggish performance, high transaction costs, and capital inefficiencies inherent to automated market makers (AMMs) operating on general-purpose blockchains.

Hyperliquid represents the resolution of this dichotomy. By engineering a bespoke Layer-1 blockchain optimized exclusively for decentralized perpetual futures and order-driven markets, Hyperliquid has redefined the frontier of on-chain trading. With sub-second finality, zero gas fees for trading, and a fully on-chain orderbook, it provides the deterministic execution and speed required by sophisticated algorithmic traders, without compromising the fundamental ethos of self-custody and transparent verifiable state.

Trading on Hyperliquid: Strategies, Market Making, and APIs for On-Chain Perps is written for the quantitative researcher, the software engineer, and the systematic trader who recognize the magnitude of this paradigm shift. Trading in this new environment requires more than just a passing familiarity with financial derivatives; it demands a rigorous synthesis of market microstructure, mathematical risk management, and distributed systems engineering. This book is designed

to bridge the gap between traditional quantitative finance and the precise technical mechanics of modern on-chain execution.

Our exploration begins with the foundational principles of perpetual futures. Before deploying capital, one must thoroughly understand the mechanics of the instruments traded. We establish a rigorous framework for navigating orderbook microstructure, leverage mathematics, and the unique risk dynamics of crypto-derivatives, ensuring that the reader possesses the mathematical intuition required to survive in highly volatile, fast-moving markets.

From these theoretical foundations, we transition deeply into the specific mechanics of Hyperliquid itself. The protocol's architecture—spanning the HyperCore matching engine, the HyperEVM, and the HyperBFT consensus model—presents unique constraints and opportunities. We deconstruct the definitive rules governing initial and maintenance margin, algorithmic liquidations, and the continuous funding rates that anchor perpetual contracts to their underlying spot prices. Understanding these native mechanics, along with the nuanced impact of maker rebates and taker fees, is the prerequisite for designing strategies with a positive expected value.

Equipped with a mastery of the venue, the text then addresses the art and science of order execution and quantitative workflow. Moving from statistical intent to realized fills is an intricate process fraught with adverse selection and slippage. We explore the tactical deployment of limit orders, protective post-only flags, and advanced time-weighted execution algorithms. Furthermore, we outline the complete quantitative research lifecycle, detailing how to construct robust market data pipelines, simulate non-deterministic execution paths with high fidelity, and attribute performance metrics accurately to isolate structural alpha from market noise.

At the heart of the text lies the formulation of systematic trading strategies. Whether you are pursuing directional momentum, structuring delta-neutral carry positions to harvest funding rates, or executing cross-venue arbitrage, the strategic frameworks provided herein are tailored to the realities of on-chain constraints. A significant portion of this material is dedicated to the complex discipline of algorithmic market making. We formally decompose the expected value of liquidity provision, offering implementable models for inventory control,

volatility-scaled quoting, and interaction with Hyperliquid’s native protocol vaults.

However, the most elegant mathematical model is entirely useless without resilient software to execute it. The final chapters of this book serve as an advanced manual for systems engineering and bot architecture. We guide the reader through the intricacies of establishing deterministic WebSocket synchronization, managing cryptographic authentication, and handling API nonces in highly concurrent environments. Beyond basic connectivity, we focus heavily on production security, emphasizing event-driven architectures, automated kill switches, latency optimization, and strict key management protocols. In a landscape where code is immediately met with financial consequence, operational resilience is just as critical as predictive accuracy.

Trading on decentralized orderbooks is an inherently multidisciplinary pursuit. It requires the mathematical rigor to price risk, the strategic acumen to navigate market microstructure, and the engineering discipline to build systems that operate flawlessly in adversarial environments. This book is your comprehensive guide to mastering that intersection. Whether you are aiming to deploy your first automated strategy or scaling an institutional market-making operation, the principles detailed in the following pages will equip you to navigate, and ultimately conquer, the Hyperliquid ecosystem.

Risk disclaimer. This book is for educational and informational purposes only and does not constitute financial, investment, tax, or legal advice, nor a recommendation or solicitation to buy or sell any asset or to adopt any trading strategy. On-chain perpetual trading, market making, and automated strategies on venues such as Hyperliquid involve substantial risk and are not suitable for every investor; leverage, liquidation, smart-contract/bridge failure, API and latency issues, and funding-regime shifts can cause rapid, total losses. Examples, figures, and code herein are illustrative and simplified, and past or hypothetical performance does not guarantee future results. Consult a licensed financial professional before trading. The author and publisher disclaim any liability for losses arising from the use of this material.

Chapter 1

Foundations of On-Chain Perpetual Futures

Before deploying automated strategies on Hyperliquid, you must fluently speak the language of on-chain derivatives. This chapter strips away the hype, grounding you in the quantitative realities of perpetual futures, orderbook microstructure, and the unique deterministic constraints of blockchain execution. Here, we build the mathematical and structural foundations necessary to engineer robust, profitable trading systems.

1.1 Perpetual Futures Contract Basics

Perpetual futures (often called *perpetual swaps* or simply *perps*) are futures-like derivatives designed to track an underlying asset price *without* an expiration date. They are the core instrument traded on perp venues: a standardized contract whose value rises when the underlying price rises (for a long) and falls when the underlying price falls (for a short). The design goal is simple: preserve the capital efficiency

and shorting convenience of futures, but remove the operational burden of rolling expiring contracts.

This section builds the minimal mental model you will use everywhere else in this book:

- what you are trading (a derivative with no maturity),
- which prices matter (index vs mark vs last),
- and why the perp price does not drift indefinitely from spot (funding as an anchoring mechanism).

1.1.1 Perps vs. traditional futures: what changes when there is no expiry?

A traditional futures contract has a maturity date. That maturity is not a superficial detail; it is the mechanism that forces convergence to spot. As expiration approaches, arbitrage and deliverability (cash-settlement or physical delivery) compress the gap between the futures price and the spot reference.

A perpetual contract removes the maturity date. That creates an immediate question:

If there is no settlement date, what prevents the perp from trading permanently above (or below) the underlying price?

The answer is **funding**. Funding replaces “expiry-based convergence” with a continuous (usually periodic) transfer between longs and shorts that makes sustained mispricing expensive to hold on the wrong side.

1.1.2 Perps vs. traditional futures: what changes when there is no expiry?

A traditional futures contract has a maturity date. That maturity is not a superficial detail; it is the mechanism that forces convergence to spot. As expiration approaches, arbitrage and deliverability (cash-settlement or physical delivery) compress the gap between the futures price and the spot reference.

A perpetual contract removes the maturity date. That creates an immediate question:

If there is no settlement date, what prevents the perp from trading permanently above (or below) the underlying price?

The answer is **funding**. Funding replaces “expiry-based convergence” with a continuous (usually periodic) transfer between longs and shorts that makes sustained mispricing expensive to hold on the wrong side.

1.1.3 The three prices you must distinguish: index, mark, and last

Even experienced off-chain futures traders can misread perp risk by using the wrong price in the wrong place. Perp venues typically maintain multiple price concepts:

- **Index price** I_t : a reference price for the underlying asset, typically derived from external spot markets. Conceptually, the index is the “best estimate” of fair spot.
- **Perp trading price** (often “last”) P_t : the most recent traded price on the perp order book.
- **Mark price** M_t : an internal, risk-engine price used for margining, liquidation logic, and (commonly) unrealized PnL. It is designed to be more manipulation-resistant than last price and to reflect the venue’s view of fair derivative value in the presence of microstructure noise.

In strategy design, you should treat these as three different signals with different failure modes:

- I_t can lag fast moves (depending on oracle construction) but is hard to manipulate locally.
- P_t is real execution, but can be noisy, especially in thin markets or during bursts.
- M_t is the venue’s “risk truth”; if your bot is managing liquidation risk, M_t is often the most operationally relevant.

1.1.4 The funding rate: the anchoring mechanism

Funding is a periodic cashflow exchanged between longs and shorts. It is not a trading fee paid to the exchange; it is a *transfer between position holders* that depends on how the perp is priced relative to the underlying reference.

Let r_f denote the funding rate applied over a funding interval. The canonical payoff logic is:

- If $r_f > 0$, **longs pay shorts**.
- If $r_f < 0$, **shorts pay longs**.

For a position with notional exposure N (in quote currency terms), the funding payment over one interval is approximately

$$\text{Funding Payment} \approx N \cdot r_f,$$

with sign depending on whether you are long or short.

Why this anchors the perp price. Funding creates a feedback loop:

- When the perp persistently trades at a *premium* to the index (perp “too expensive”), the mechanism typically produces $r_f > 0$. Longs now pay to keep the position. That discourages holding longs and encourages holding shorts, pushing the perp price downward toward the index.
- When the perp trades at a *discount* to the index (perp “too cheap”), the mechanism tends to produce $r_f < 0$. Shorts pay to keep the position. That discourages shorts and encourages longs, pushing the perp price upward toward the index.

A useful way to think about funding is as a **carrying cost** applied to the basis (the perp–index gap). The basis is

$$b_t = P_t - I_t$$

or, in some contexts,

$$b_t = M_t - I_t.$$

The exact implementation differs by venue, but the economic intent is stable: the larger and more persistent the basis, the larger the incentive to take the opposite side and earn funding.

1.1.5 Funding is not “free yield”: the basis-risk trade

A common beginner mistake is to interpret positive funding as guaranteed profit for shorts (or negative funding as guaranteed profit for longs). In reality, funding is only one leg of a combined trade:

- You earn/pay funding based on your perp position.
- Simultaneously, you take price risk on the perp, and often basis risk between the perp and the underlying reference.

Even a *delta-neutral* funding strategy (e.g., short perp, long spot) is not risk-free. Key frictions:

- **Basis volatility:** the perp can move relative to spot/index, producing mark-to-market swings that can exceed funding earned.
- **Execution and financing:** entering both legs has spread, slippage, and sometimes borrow or capital costs.
- **Liquidation mechanics:** perps are leveraged instruments; adverse basis moves can force liquidation even if the long-run funding thesis is correct.

The practical consequence for bot design is that funding must be modeled as a *cashflow stream* inside a risk-managed strategy, not as a deterministic edge.

1.1.6 A minimal accounting model for perp economics

For a single perp position, your PnL over a horizon can be decomposed into:

$$\text{Total PnL} \approx \text{Price PnL} + \text{Funding PnL} - \text{Trading costs.}$$

Price PnL (conceptual). If you hold position size q (in base units, positive for long, negative for short), then a simple approximation is:

$$\text{Price PnL} \approx q \cdot (M_{t_1} - M_{t_0})$$

when the platform marks positions to M_t . The exact platform accounting may separate realized vs unrealized PnL; the point here is the dependency on the mark.

Funding PnL (conceptual). Across funding intervals $k = 1, \dots, K$ with rates $r_{f,k}$ applied to your relevant notional N_k , the funding contribution is approximately

$$\text{Funding PnL} \approx \sum_{k=1}^K (-\text{sgn}(q)) N_k r_{f,k},$$

where the sign convention matches “longs pay when $r_f > 0$.”

This decomposition is the foundation for everything later: market making requires modeling adverse selection in price PnL; carry strategies require modeling variance in both price PnL and funding PnL; and robust risk management requires stress testing the combined distribution.

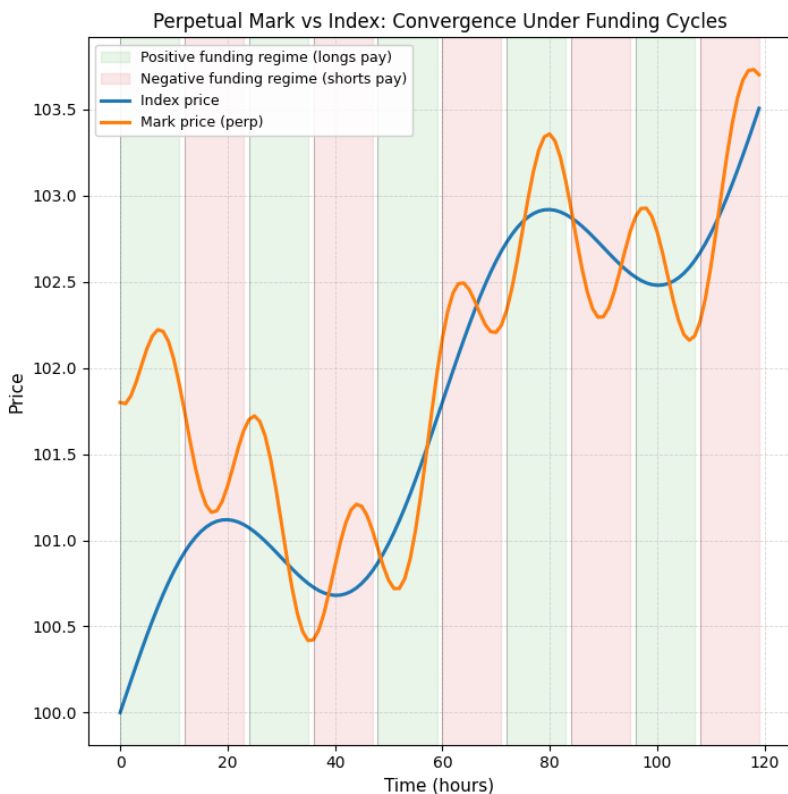
1.1.7 Funding cycles and convergence intuition

Because funding is applied discretely in time (e.g., every T hours), the anchoring force is also discrete. That leads to an observable pattern in many perp markets:

- The perp price can drift away from the index during aggressive directional flow.
- As funding payments approach and/or as funding rates rise in magnitude, the cost of holding the “expensive” side increases.
- Arbitrage and positioning gradually compress the basis, especially when balance sheet and execution allow it.

This is not a law of nature; it is a behavioral and capital-allocation response to a designed incentive. In stressed markets, basis can remain

wide because risk limits, liquidity, and liquidation constraints dominate.



1.1.8 What a perp contract gives you operationally

For on-chain traders and quants, perps are attractive because they package several capabilities into one instrument:

- **Efficient directional exposure:** you can express bullish or bearish views without borrowing the underlying asset.

- **Leverage with continuous risk checks:** margining allows larger notional exposure than posted collateral, but introduces liquidation risk that is path-dependent.
- **A natural “carry” dimension:** funding makes the cost/benefit of holding a position explicit and time-varying, which can be modeled and traded.

These benefits are exactly why the remainder of the chapter will emphasize (i) order book microstructure, because it determines your realized entry/exit prices, and (ii) position and margin mathematics, because leverage turns small pricing and basis errors into discontinuous outcomes.

1.1.9 Checklist: the perp basics your bot must get right

Before writing a single line of trading logic, ensure your system can answer these questions for every market you trade:

- Which price does your strategy use for signals (I_t , P_t , M_t), and which price does the venue use for risk and liquidation?
- How often is funding applied, and what is the sign convention (who pays whom when funding is positive)?
- Does your expected edge come from price movement, funding, or both—and what is the distribution of basis moves that can overwhelm funding?
- Are you measuring PnL in a way consistent with how the venue will realize losses (mark-based liquidation, fees, and funding debits/credits)?

If any of these are ambiguous, you do not yet have a tradable model; you have a backtest artifact.

1.2 Orderbook Market Microstructure

The previous section established *what* a perpetual contract is and *why* its price is anchored to an external reference via funding. This section explains *how* you actually trade that contract in an order-driven market: a **central limit order book (CLOB)**. For systematic traders, market making is not primarily a *directional* problem; it is a problem of *microstructure*: spreads, depth, queue priority, adverse selection, and market impact.

1.2.1 The CLOB: two stacks of conditional liquidity

A CLOB is a matching engine organized around two sets of *resting* limit orders:

- **Bids**: buy orders specifying a maximum price the trader is willing to pay.
- **Asks**: sell orders specifying a minimum price the trader is willing to accept.

A limit order is *conditional liquidity*: it provides liquidity only if the market trades to its price. A market order is *immediate liquidity demand*: it consumes the best available resting liquidity until its size is filled.

Two basic definitions drive much of your execution math.

Best bid b_0 is the maximum of the bid prices. Best ask a_0 is the minimum of the ask prices.

The **bid–ask spread** is

$$s = a_0 - b_0,$$

and the **mid-price** (a common reference for quoting and inventory control) is

$$m = \frac{a_0 + b_0}{2}.$$

Spreads are not arbitrary. In microstructure terms, spreads compensate liquidity providers for (i) order processing costs, (ii) inventory risk, and (iii) adverse selection risk. These three components vary across time: in calm conditions, spreads can be tight because inventory and information risks are lower; during volatility or information events, spreads typically widen because providing liquidity becomes more dangerous.

1.2.2 Depth and slippage: why size changes your price

The spread tells you the cost of trading *a tiny amount*. Real strategies must trade nontrivial size, which brings in **depth**: how much quantity is available at each price level.

A convenient way to think about depth is as a step function: at each price level you can trade up to the resting quantity there; if you need more, you must trade at worse levels. When a market order “walks the book,” the average fill price moves away from the mid-price. That difference is **slippage** (more generally, part of execution cost).

A toy example (prices in quote currency, quantities in base units) makes the mechanics concrete:

Ask Qty	Ask Price	Bid Price	Bid Qty
0.7	100.04	100.00	1.1
1.0	100.06	99.98	1.4
1.8	100.10	99.95	2.2

If you submit a market buy for $Q = 1.2$ units, you would typically fill:

- 0.7 at 100.04 (best ask),
- and the remaining 0.5 at 100.06 (next ask level).

The volume-weighted average fill price becomes

$$\text{VWAP} = \frac{0.7 \cdot 100.04 + 0.5 \cdot 100.06}{1.2},$$

which is worse than the best ask. This is not a “fee”; it is the mechanical cost of demanding immediacy in a finite-depth book.

1.2.3 Price–time priority and queue position

A CLOB matches orders using a priority rule. The most common is **price–time priority**:

- Better price executes first (higher bids, lower asks).
- For identical prices, earlier orders execute first.

This rule creates the concept of **queue position**. If you post a limit order at the best bid, you are joining a queue of existing orders at that same price. Your fill probability depends on (i) how much volume arrives to trade against that price, and (ii) how many units are in front of you.

Queue position is a central driver of market-making behavior:

- If you continuously cancel/replace to chase the best price, you may reset your time priority and fall to the back of the queue.
- If you never adjust, you may keep priority but become stale relative to the true price, increasing adverse selection risk.

The tension between *being first* and *being right* is where much of practical market making lives.

1.2.4 Spread decomposition: why market making is not “set and forget”

A spread is compensation for providing immediacy to others. Microstructure theory organizes the spread into three intuitive components:

Order processing and operational costs. Even in frictionless theory, a venue has fees and a market maker has infrastructure costs. At the strategy level, you can treat this as a minimum spread needed to avoid negative expectancy after fees.

Inventory risk (the dealer problem). If you provide both bids and asks, fills arrive asymmetrically. You can accumulate inventory (net long or net short). If the price trends against your inventory, you lose.

Inventory risk increases with volatility and with the speed at which you can rebalance. This is why many market makers widen quotes when volatility rises: wider quotes reduce fill probability and increase per-fill compensation.

Adverse selection (the informed-trader problem). Glosten–Milgrom (1985) formalizes a simple but powerful idea: some order flow is information-driven. When a trader aggressively buys, it may be because the true value is higher than the current quotes. If you sell to that trader at the ask, you are trading *against* information and your expected PnL is negative unless you widen the spread (or otherwise adjust) to compensate.

In practice, adverse selection is why market-making performance often deteriorates *exactly when volume increases*. More trades do not automatically mean more profit; they can mean more informed trading.

1.2.5 Price impact and the Kyle lens

Kyle (1985) provides a canonical model linking order flow to prices through market depth: larger net buy pressure pushes prices up, and larger net sell pressure pushes prices down. While the Kyle model is stylized, its core message is operationally essential:

Execution cost grows with size because your own trading moves the price.

For strategy design, you can separate impact into two parts:

- **Temporary impact:** the immediate concession you pay to access liquidity (walking the book, crossing the spread).
- **Permanent impact:** the information your order flow reveals (or is inferred to reveal), shifting the market’s estimate of fair value.

A practical market maker tries to avoid being the “free option” for informed traders: if aggressive flow arrives that is likely information-driven, you want your quotes to either (i) step away (widen), (ii) skew (quote less aggressively on the side that would worsen your inventory), or (iii) reduce size.

1.2.6 Microstructure metrics you will reuse in systematic market making

To implement market making as an engineering system, you need measurable quantities that map to the concepts above:

Top-of-book and spread. Track (b_0, a_0) and s . Spreads typically widen in volatility and thin liquidity.

Order book imbalance. A simple signal compares bid-side depth near the top vs ask-side depth near the top. Persistent imbalance can predict short-horizon price moves, but it can also be a trap if imbalance reflects stale liquidity that will cancel.

Depth profile and liquidity holes. Not all depth is equal: you care about depth near the top for immediate execution, and depth further out for tail slippage during stress. Liquidity holes (missing levels) can create discontinuous slippage.

Queue position and expected fill probability. For a posted limit order at price p , define $Q_{ahead}(p)$ as the resting quantity ahead of you at p . If expected opposing marketable volume over your holding horizon is V , a rough first approximation is that your fill probability increases as V exceeds $Q_{ahead}(p)$ (with many caveats due to cancellations and partial fills). This is why time priority matters and why cancellations by others can be as important as trades.

Short-horizon adverse selection proxy. A common empirical proxy is the *markout*: after you get filled, measure the price change over the next few seconds. If you sell and the price immediately rises, you were adversely selected. Market makers build rules to minimize negative markouts.

1.2.7 Why microstructure is the foundation for automated liquidity provision

Once you adopt the microstructure lens, market making becomes a controlled trade-off:

- Tighter quotes increase fill probability but increase adverse selection risk.

- Wider quotes reduce adverse selection but reduce trading frequency and can increase inventory drift.
- Aggressive cancel/replace can keep you close to fair value but can sacrifice time priority and increase missed fills.

The remainder of this book will build systematically on these mechanics. Position and leverage mathematics determine how microstructure losses translate into liquidation risk; risk metrics quantify tail outcomes when liquidity vanishes; and API-driven execution turns these concepts into real-time quoting, cancellation, and inventory control.

1.3 Position, PnL, and Leverage Mathematics

The previous sections established (i) how perps are economically anchored through funding and (ii) how the order book determines your realized execution prices. This section turns those mechanics into accounting identities you can implement in code: how to represent a position, compute notional exposure and leverage, decompose PnL into price and funding components, and understand why liquidation distance shrinks rapidly as leverage rises.

Throughout, we will keep the mathematics venue-agnostic where possible, while respecting two perp-specific realities: (1) PnL is driven by both price changes and funding accrual, and (2) liquidation logic typically uses the mark price (as previously defined) rather than the last traded price.

1.3.1 State variables: the minimal position record

An automated trading system should maintain a clean, explicit state for each market i it trades. A minimal record is:

- Position size in base units: q_i (positive long, negative short).
- Average entry price: $\bar{p}_{i,entry}$ (in quote per base).

- Current mark price: M_i .
- Cumulative funding paid/received: F_i (in quote currency).
- Fees paid: Φ_i (in quote currency).

In addition, risk checks require account-level state:

- Account equity: E .
- Used margin: U .
- Available (free) margin: $A = E - U$.

Exactly how E and U are computed depends on the margin model (isolated vs cross) and venue-specific haircuts and buffers. The bot engineering principle is universal: *separate what you observe from what you infer*. If the platform provides equity and margin summaries via API, prefer those as the source of truth and use your local math as a reconciliation check.

1.3.2 Notional exposure and leverage

For a perp position, the **notional** is the dollar (quote) value of exposure:

$$N_i = |q_i| \cdot M_i.$$

Notional is not the same as margin. Notional is the size of the bet; margin is the collateral posted to support that bet.

Define account equity E as the current value of your collateral plus unrealized PnL (and typically including funding accruals, depending on settlement conventions). Then a common definition of **leverage** at the account level is:

$$L = \frac{\sum_i N_i}{E}.$$

For isolated margin, one also speaks of per-position leverage:

$$L_i = \frac{N_i}{m_i},$$

where m_i is the margin allocated to position i .

Two operational implications.

- If E drops (losses, funding paid, fees), then leverage rises even if notional stays constant. This is why a bot should monitor leverage continuously, not only at order placement time.
- If M_i rises while you are long, notional increases. That can increase leverage and reduce liquidation distance despite the position being profitable. This surprises many traders: profit can increase risk if it increases notional faster than equity.

1.3.3 Mark-to-market: price PnL for a single position

Classical futures mechanics are built around mark-to-market: the position is revalued continuously (or discretely at settlement times) against a reference price. In perp venues, unrealized PnL is typically tied to the mark price rather than last trade.

For a position with size q and average entry price $\bar{p}_{entry}$, a basic **unrealized price PnL** model is:

$$uPnL_{price} = q (M - \bar{p}_{entry}).$$

This sign convention works naturally:

- If $q > 0$ (long) and M rises above entry, $uPnL_{price}$ is positive.
- If $q < 0$ (short) and M falls below entry, $uPnL_{price}$ is positive.

Realized vs unrealized. When you reduce or flip a position (sell some of a long, buy some of a short), part of the PnL becomes realized. A clean implementation keeps:

$$\text{Total PnL} = \text{Realized PnL} + \text{Unrealized PnL}.$$

Your bot should treat realized PnL as cash-like (it changes equity immediately), while unrealized PnL is mark-dependent and can change each tick.

1.3.4 Funding accrual: PnL when the price does not move

Perps add a second PnL stream: **funding**. As discussed earlier, funding is a transfer between longs and shorts. The key accounting point is that funding changes equity even if price is flat.

Let funding be applied at discrete times t_k with funding rate r_k for the interval. A simple funding cashflow model for a position is:

$$\Delta F_k = -\text{sgn}(q) N_k r_k,$$

where $N_k = |q| M_{t_k}$ is the notional at the funding timestamp, and the sign captures “longs pay when funding is positive.”

Then cumulative funding PnL is:

$$F = \sum_{k=1}^K \Delta F_k.$$

Example: positive funding with a flat market. Assume:

- You are short $q = -2$ units.
- Mark price is constant at $M = 100$.
- Funding is $r = +0.01\% = 0.0001$ per interval for $K = 10$ intervals.

Notional per interval is $N = |-2| \cdot 100 = 200$. Since $q < 0$, $-\text{sgn}(q) = +1$, so:

$$\Delta F = 200 \cdot 0.0001 = 0.02, \quad F = 10 \cdot 0.02 = 0.2.$$

Your price PnL is near zero, but your equity increases by 0.2 (ignoring fees). The symmetric statement holds for longs: positive funding is a cost of holding longs, negative funding is a cost of holding shorts.

1.3.5 Total PnL decomposition (what you should log)

For each symbol, it is good practice to log PnL as:

$$PnL_i = rPnL_{i,price} + uPnL_{i,price} + F_i - \Phi_i.$$

At the account level, equity E equals Collateral Value plus

$$\sum_i uPnL_{i,price} + \sum_i rPnL_{i,price} + \sum_i F_i - \sum_i \Phi_i.$$

Platforms may apply realized PnL, funding, and fees in specific settlement steps. The engineering requirement is to reconcile your computed E to the venue-reported E ; any persistent discrepancy is a risk bug.

1.3.6 Margin, initial vs maintenance, and why liquidation is mark-based

Classical futures mechanics (as in traditional clearing) distinguish:

- **Initial margin (IM):** collateral required to open/extend positions.
- **Maintenance margin (MM):** minimum collateral required to keep positions open.

If equity falls below maintenance margin, a margin call (or forced reduction/liquidation) occurs.

Perp venues implement the same economic idea, but operationally they often:

- mark positions using the **mark price** (not last trade),
- and compute liquidation thresholds from mark-based equity and margin usage.

Using the mark price for liquidation reduces the incentive to manipulate the last traded price with small, aggressive trades. For an automated trader, the key takeaway is: *your liquidation risk is tied to M , not your preferred chart price.*

1.3.7 Liquidation intuition: leverage turns small moves into discontinuous outcomes

A full liquidation model is venue-specific, but the intuition is universal and worth formalizing.

Consider a single isolated position with margin m and notional N . Define leverage $L = N/m$. If the market moves against you by a fraction δ (e.g., -1% for a long), the approximate loss is $N\delta$. As a fraction of your margin, that is:

$$\frac{\text{Loss}}{m} \approx \frac{N\delta}{m} = L\delta.$$

So:

- at $L = 1$, a 1% adverse move costs about 1% of margin;
- at $L = 20$, a 1% adverse move costs about 20% of margin.

Liquidation occurs when remaining equity approaches maintenance margin (plus any buffers). This makes liquidation distance shrink roughly in proportion to $1/L$ once fees and maintenance requirements are included. The qualitative point is what matters for strategy design: **leverage converts ordinary volatility into liquidation probability.**

1.3.8 Cross margin vs isolated margin: why liquidation thresholds can drift

In isolated margin, each position has its own margin bucket. In cross margin, positions share a common equity pool. This changes liquidation dynamics in a way bots must handle carefully.

Isolated margin (single-position mental model). Risk is localized:

- Position i can be liquidated without necessarily impacting other positions (beyond any shared collateral mechanisms the venue might apply).
- Liquidation price depends mainly on q_i , M_i , $\bar{p}_{i,entry}$, and m_i .

Cross margin (portfolio mental model). Risk is coupled:

- One position's unrealized losses can consume shared equity and endanger other positions.
- Funding payments and unrealized PnL in other markets can change your liquidation threshold for a given symbol, even if that symbol's mark price is unchanged.

This is why, operationally, liquidation thresholds can drift over time. If the venue reports a liquidation price estimate, treat it as a dynamic account-level risk metric, not a fixed property of a position.

1.3.9 Continuous monitoring: what an automated system should observe

The most robust design pattern is:

- Use the venue API to fetch the current state required for risk checks (mark price, funding, open interest, liquidation price estimate, margin summary).
- Independently compute local estimates (notional, leverage, PnL decomposition).
- Reconcile and alert on discrepancies.

Automated systems can observe required state variables (mark price, funding, open interest, liquidation price estimate, margin summary) via the venue's API endpoints. If you cannot obtain a variable deterministically from the API, do not build a safety mechanism that depends on it; instead, build conservative buffers using what you can observe.

1.3.10 Implementation checklist (math you should unit-test)

A bot that trades perps should have unit tests for each of the following:

- **Sign conventions:** long/short sign for q , and funding payment direction when $r_f > 0$.
- **Notional and leverage:** $N = |q|M$ and $L = \sum N/E$ update correctly as M and E change.
- **PnL decomposition:** realized/unrealized price PnL and funding accrual sum to total PnL, net of fees.
- **Cross-coupling effects:** in cross margin, changing PnL or funding in one symbol changes account equity and therefore leverage and liquidation distance for all symbols.

These are not accounting details. They are the difference between a bot that opportunistically trades a spread and a bot that survives regime changes. The next section will build on this foundation by formalizing risk metrics (volatility, drawdown, and tail-event safeguards) that translate these accounting variables into automated limits and defensive logic.

1.4 Core Risk Metrics for Traders and Quants

The previous section converted perp mechanics into implementable accounting: positions, notional, leverage, and mark-based PnL (including funding). Risk management starts where accounting ends. Once you can measure exposure and equity correctly, you can impose *constraints* that keep an automated strategy alive through volatility spikes, liquidity droughts, and discontinuous moves.

Perp risk is structurally **tail-sensitive**: leverage means small adverse moves can cause discontinuous outcomes (forced liquidation). In addition, mark-price-based liquidation rules mean your risk metrics should be computed versus *mark/index dynamics* (basis plus volatility), not merely last-trade volatility.

This section introduces the metrics and control rules that are most useful in practice for on-chain perp bots:

- volatility and variance (to normalize risk across regimes),
- drawdown (to control path-dependent capital destruction),
- tail-risk measures (VaR vs Expected Shortfall/CVaR),
- and operational safeguards (limits, buffers, circuit breakers, and stress tests).

1.4.1 Returns, variance, and volatility: the baseline units of risk

Let E_t be your account equity (marked using the venue's risk price, typically the mark price for positions). Define discrete-time returns:

$$r_t = \frac{E_t - E_{t-1}}{E_{t-1}}.$$

For higher-frequency systems where E_t changes continuously, log returns are often convenient:

$$\ell_t = \log\left(\frac{E_t}{E_{t-1}}\right).$$

The **variance** of returns over a window of n observations is

$$\hat{\sigma}^2 = \frac{1}{n-1} \sum_{t=1}^n (r_t - \bar{r})^2, \quad \bar{r} = \frac{1}{n} \sum_{t=1}^n r_t,$$

and **volatility** is $\hat{\sigma} = \sqrt{\hat{\sigma}^2}$.

Why bots need volatility estimates. Raw PnL or raw equity changes are not comparable across regimes. A 0.5% equity move might be a daily tail event in calm markets and a routine fluctuation during a liquidation cascade. Volatility converts PnL and liquidation distance into standardized units so that a risk rule can scale exposure up/down coherently.

Mark vs last: what you should compute volatility on. Because liquidation logic is mark-based, and because perps can trade away from the index during stress, your most relevant short-horizon risk is the volatility of:

- your equity E_t marked to the platform's mark price, and
- the basis component $M_t - I_t$ when you run basis-sensitive strategies (market making, funding carry, hedged spot-perp books).

1.4.2 Maximum drawdown: a path-dependent risk that traders actually feel

Variance and volatility are symmetric: they penalize upside and downside equally. Traders care more about losses, and in leveraged perps the *path* of losses matters because a temporary drawdown can force liquidation even if the long-run expectancy is positive.

Define the running peak equity:

$$P_t = \max_{0 \leq \tau \leq t} E_\tau.$$

Define drawdown at time t as

$$D_t = \frac{P_t - E_t}{P_t}.$$

Then the maximum drawdown over a horizon is

$$\text{MDD} = \max_t D_t.$$

Why MDD is operationally useful. MDD answers a concrete question: “How bad did it get from peak to trough?” This maps naturally to human and systematic risk budgets:

- If your strategy has a 25% MDD in backtests, it will frequently be at risk of hitting liquidation or risk-off thresholds at moderate leverage.
- Two strategies can have similar volatility but very different MDD because one has clustered losses (serial correlation) and the other does not.

1.4.3 Tail events and gap risk: why stop-loss logic can fail

A **tail event** is an extreme outcome far in the distribution tail: large adverse moves that occur rarely but dominate risk. In perps, tail events couple with liquidity and execution constraints:

- spreads widen,
- depth disappears,
- mark/index basis can widen,
- and your exit orders may fill at much worse prices than expected.

Gap risk is a specific failure mode where the price (or mark) moves discontinuously, “jumping” over your intended stop level. In a CLOB, a stop is not a guarantee of execution at the stop price; it is a *trigger* that converts into a marketable order, which then fills against the available book. If the book is thin or moving fast, the realized fill can be far from the trigger.

The robust way to design risk controls is to assume that *your worst fills occur exactly when you most need to trade*.

1.4.4 Distance to liquidation as a risk budget

Because liquidation is discontinuous, a highly practical risk metric is the **distance to liquidation**. If the venue provides a liquidation price estimate, you can convert it into a percentage move from the current mark price. For long and short positions define:

$$d_{liq}^{long} = \frac{M - M_{liq}}{M}, \quad d_{liq}^{short} = \frac{M_{liq} - M}{M}.$$

These produce single numbers with direct meaning: the adverse mark move (in %) that would liquidate you, ignoring second-order effects.

Normalize liquidation distance by volatility. To make this regime-aware, express the liquidation distance in “sigma units.” Let

$\hat{\sigma}_M$ be an estimate of mark-price volatility over your relevant horizon. Then:

$$z_{liq} \approx \frac{d_{liq}}{\hat{\sigma}_M}.$$

If z_{liq} is small (e.g., ≈ 2), ordinary noise can liquidate you. If z_{liq} is larger (e.g., ≈ 8), liquidation requires an extreme event. This is not a guarantee—tails are not Gaussian—but it is an actionable normalization.

Why mark/index dynamics matter. For a leveraged market maker or a basis-sensitive strategy, liquidation can be driven by basis widening even if the underlying spot is stable. That is why monitoring both M_t volatility and basis volatility (e.g., $M_t - I_t$) is part of robust risk control.

1.4.5 VaR and Expected Shortfall (CVaR): measuring tails correctly

Value-at-Risk (VaR) is a common tail metric: for confidence level α (e.g., 95%), VaR_α is a loss threshold such that losses exceed it only $1 - \alpha$ of the time. In return terms, if L is the loss random variable over a horizon:

$$\text{VaR}_\alpha = \inf\{\ell : P(L \leq \ell) \geq \alpha\}.$$

VaR has a major limitation: it does not tell you how bad losses are beyond the threshold. For leveraged perps, the worst outcomes dominate survivability, so you often want **Expected Shortfall** (also called CVaR):

$$\text{ES}_\alpha = E[L \mid L \geq \text{VaR}_\alpha].$$

Why quants prefer ES/CVaR for tails. Coherent risk measures (Artzner et al., 1999) propose axioms for a “well-behaved” risk measure; a key one is **subadditivity**, which formalizes diversification: risk of a combined portfolio should not exceed the sum of standalone risks. VaR can violate subadditivity in some distributions, while Expected Shortfall satisfies it under broad conditions. In practice, this means ES is generally a better tool for leveraged books with nonlinear liquidation effects and fat tails.

1.4.6 Stress tests: the risk tool that does not lie

Any metric estimated from recent history can fail when the regime changes. Stress tests address this by asking: “What happens if the world looks *worse* than my window suggests?”

A useful perp stress test set includes:

- **Mark move stress:** apply $\pm x\%$ shocks to M and recompute equity, leverage, and liquidation distance.
- **Basis widening stress:** shock $(M - I)$ by a widening amount, which is especially relevant during dislocations.
- **Liquidity stress:** assume your exit fills y bps worse than current top-of-book, or that you must walk multiple levels (execution slippage).
- **Funding stress:** apply an adverse funding rate for several intervals (important for carry-heavy strategies).

Stress testing is where microstructure meets risk: a “5% price shock” is not just a PnL hit; it changes spreads, fill probabilities, and liquidation odds.

1.4.7 Operational safeguards for automated strategies

Metrics are only useful if they map to *actions*. For bots, risk management is a set of deterministic rules that reduce exposure when predefined thresholds are breached.

A robust baseline control set is:

Position and notional limits.

- Max notional per instrument: $N_i \leq N_{i,\max}$.
- Max gross notional: $\sum_i N_i \leq N_{\max}$.
- Max concentration: $\max_i N_i / \sum_j N_j \leq c_{\max}$.

Leverage limits that tighten in volatility. A static leverage cap is better than nothing, but a volatility-aware cap is safer:

$$L_{\max}(t) = \min\left(L_0, \frac{K}{\hat{\sigma}(t)}\right),$$

where K is a risk budget constant and $\hat{\sigma}(t)$ is a rolling volatility estimate. The exact functional form is less important than the principle: *leverage should fall when volatility rises.*

Drawdown-based risk-off. Define a maximum allowable draw-down $D_{\max}$; if $D_t > D_{\max}$, reduce risk by:

- canceling passive quotes (for market makers),
- closing noncore positions,
- and entering a “cooldown” period where new risk is not added.

Liquidation buffer. Maintain a minimum liquidation distance:

$$d_{liq} \geq d_{min}.$$

If d_{liq} falls below the buffer, the bot must reduce exposure regardless of alpha signals. This is a hard safety constraint, not an optimization target.

Circuit breakers for tail events. Trigger a hard stop in extreme conditions, such as:

- sudden volatility spike beyond a threshold,
- repeated failed order placements/cancellations,
- or rapid basis widening beyond historical bounds.

Circuit breakers are not about maximizing profit; they are about preventing catastrophic failure modes during transient market or infrastructure instability.

1.4.8 A practical risk dashboard: what to compute every tick

Even without naming specific API identifiers, a trading bot can and should poll the venue for the state required to compute:

- Equity E and used/available margin (U , A).
- Per-market mark price M_i , and index where relevant.
- Funding rate and next funding time (for carry and PnL attribution).
- Open interest (as a regime/liquidity indicator).
- Liquidation price estimate and liquidation distance d_{liq} .
- Rolling vol estimates on equity and key marks, plus basis volatility for dislocation-sensitive strategies.
- Current drawdown D_t and max drawdown over relevant horizons.

The implementation lesson is consistent with the earlier accounting section: rely on venue-reported risk state when available, compute local mirrors for validation, and treat any discrepancy as a production incident.

1.5 Blockchain Execution Principles

So far we have treated trading as if the matching engine were an abstract machine: you submit orders, you receive fills, your positions are marked, and risk metrics update. On-chain perps add a second layer of constraints: *the execution path itself*. Even if the perp contract and the order book are well understood, a bot can still fail because of latency, inclusion uncertainty, or adversarial transaction ordering.

This section develops a practical mental model for on-chain execution. The emphasis is on what is *deterministic* (state transitions once finalized), what is *probabilistic* (ordering and inclusion until finality), and how to price these effects into execution and risk controls.

1.5.1 Deterministic state transitions vs. uncertain ordering

Blockchains give you a strong guarantee that traditional electronic markets cannot: once a block is finalized, the state is *replicated and verifiable*. A trade is not just “reported”; it is part of a public state machine.

However, before finality, two uncertainties matter for traders:

- **Inclusion uncertainty:** will your transaction be included soon, or delayed?
- **Ordering uncertainty:** if multiple transactions touch the same market state (orders, cancels, margin transfers), in what order will they be applied?

These uncertainties create a gap between the strategy you *intended* (e.g., cancel old quote, place new quote) and the strategy the chain *actually executes* (e.g., place executes before cancel, or both are delayed into a worse price regime).

1.5.2 Finality and reorg profile: why HyperBFT matters

Hyperliquid states it is secured by **HyperBFT** (a HotStuff-variant BFT). The practical takeaway is that a BFT-style consensus typically implies a different finality and reorg profile than probabilistic-finality chains. For trading system design, that impacts:

- **When you treat a fill as final:** in a fast-finality environment, you can reconcile positions and risk more aggressively after confirmation.
- **How you reason about rollback risk:** probabilistic-finality systems require thinking in terms of confirmation depth; BFT-style finality tends to offer more deterministic commitment once finalized.

Even with strong finality, do not confuse *deterministic state transitions* with *deterministic ordering*. Consensus can finalize a block de-

terministically while still allowing competition over which transactions make it into that block and in what order.

1.5.3 Latency decomposition: where time is spent

For on-chain trading, “latency” is not one number. It is the sum of several components:

- **Local processing time:** signal computation, risk checks, signing.
- **Network propagation:** getting your transaction from your machine to an ingress/mempool/leader.
- **Inclusion delay:** waiting until a block producer includes your transaction.
- **Finality delay:** waiting until the network considers the block final.
- **State observation delay:** your data feed/API view updates after inclusion/finality.

A good trading system measures each component separately. If you only observe “time from decision to fill,” you cannot tell whether you are losing because your strategy is slow, your network path is slow, or your transaction is not being prioritized.

Why latency matters more in an order book. In a CLOB, being late is a two-sided penalty:

- If you are taking liquidity, you pay more slippage because the book has moved.
- If you are providing liquidity, you become stale and are more likely to be adversely selected.

These are exactly the microstructure costs discussed earlier, but amplified by the time it takes for your intent to become final state.

1.5.4 Transaction ordering and MEV: adversarial execution is part of the game

MEV research demonstrates that transaction ordering can be adversarial: bots compete for priority, and reordering/insertion/censorship can extract value from ordering-dependent protocols. For an on-chain perp trader, the lesson is not academic. It means you must assume:

- Your transaction can be placed *before* or *after* other traders' transactions that you did not see at decision time.
- The economic value of reordering exists whenever two actions are state-dependent (e.g., cancel vs place, reduce-only close vs liquidation, hedge leg A vs hedge leg B).

A concrete ordering hazard: cancel-then-place vs place-then-cancel. Suppose your strategy maintains a tight quote and wants to move it up by one tick:

Intent: Cancel old ask at p $\rightarrow$ Place new ask at $p + \Delta$.

If ordering is unfavorable or inclusion is delayed, the chain could execute:

Place new ask $\rightarrow$ Cancel old ask.

You briefly have *two* asks posted, doubling your exposure on that side. In calm markets this may be harmless; in fast moves it can create unexpected inventory and liquidation risk.

The opposite ordering can also be harmful:

Cancel old ask $\rightarrow$ (delay) $\rightarrow$ Place new ask.

Now you temporarily have *no* offer posted, losing queue position and missing fills. The key point is that deterministic state transitions do not eliminate ordering risk; they just make the realized ordering observable after the fact.

1.5.5 Fee markets and inclusion probability

Fee-market design influences inclusion probability and ordering games. EIP-1559-style mechanisms separate base fee and tips and

change the dynamics of paying for priority and estimating costs. While the exact fee mechanism depends on the chain, from a trader’s perspective the engineering requirement is stable:

- **Model inclusion as stochastic:** your transaction has a distribution of inclusion times.
- **Treat priority as a price:** if you need timely execution, you pay more (explicitly or implicitly) to reduce delay risk.

For execution-sensitive strategies (tight spread capture, short-horizon alpha), it is often rational to pay for priority during volatile intervals because the cost of delay is larger than the incremental fee. For passive strategies, paying for priority may be negative expectancy because the edge per trade is smaller.

1.5.6 A chain-aware execution cost model

$$\mathbb{E}[\text{Total Cost}] \approx \underbrace{\text{Spread}}_{\text{crossing}} + \underbrace{\text{Slippage/Impact}}_{\text{walking the book}} + \underbrace{\text{Fees}}_{\text{explicit}} + \underbrace{\text{Delay Cost}}_{\text{latency exposure}} + \underbrace{\text{Ordering/MEV}}_{\text{adversarial or}}$$

Delay cost as “exposure during transit”. If your transaction is in flight for Δt , you are exposed to price movement during that time. A first-order approximation is:

$$\text{Delay Risk} \propto \sigma \sqrt{\Delta t},$$

where σ is a volatility estimate for the relevant price (often the mark price). This is not a precise pricing formula; it is a design heuristic that explains why the same strategy can be profitable in low-volatility regimes and unprofitable when volatility rises or inclusion slows.

Ordering/MEV premium as a worst-case adjustment. If the PnL of your action depends strongly on being early (e.g., canceling a stale quote before informed flow arrives), then you should treat adversarial ordering as an implicit tax. The correct response is usually not to “outbid everyone,” but to:

- reduce reliance on fragile sequences,

- widen or skew quotes when volatility rises,
- and increase buffers so that brief ordering inversions do not threaten solvency.

1.5.7 Engineering patterns: making execution robust

Execution robustness is mostly about avoiding brittle assumptions.

Idempotent intent and reconciliation. Design order actions so that resubmitting the same intent does not create uncontrolled exposure. Then reconcile observed open orders and positions to your internal desired state at a fixed cadence:

- If an order is missing, replace it.
- If an order exists that should not, cancel it.
- If position differs from expected (partial fills, delayed cancels), update inventory and risk immediately.

Avoid long dependency chains. The more sequential your actions (A must happen before B must happen before C), the more you are exposed to ordering uncertainty. Prefer designs where a single action moves you to a safe state, even if it is not optimal. For example, in stress you might choose to reduce inventory with a single marketable order rather than juggling many cancel/replace steps.

Conservative exposure under uncertainty. When you cannot be sure of ordering, you must budget for temporary worst cases:

- Assume you may have both old and new quotes live briefly.
- Assume a hedge leg may be delayed relative to the first leg.
- Assume stops can gap and fill worse than intended (as discussed in risk metrics).

This is the execution-layer analogue of a liquidation buffer: it is a margin of safety for sequencing risk.

1.5.8 Summary: what on-chain execution changes for traders

On-chain perps combine an order book with a blockchain execution layer. The blockchain gives determinism after finality and auditability of outcomes, but it introduces inclusion and ordering uncertainty before finality. Hyperliquid's HyperBFT security model implies a distinct finality profile that influences how quickly you can treat state as final. MEV research makes clear that ordering can be adversarial, so strategy design must tolerate reordering and delay. Finally, fee-market dynamics translate urgency into a measurable cost.

In the chapters that follow, these principles will be used to justify concrete bot design choices: how frequently to cancel/replace, how to set volatility-aware quote widths, how to implement risk-off modes, and how to reconcile state so that your internal model cannot drift away from the chain's reality.

Chapter 2

Hyperliquid Architecture and Market Mechanics

Enter the engine room of Hyperliquid. For quantitative traders and market makers, alpha is inextricably linked to infrastructure. This chapter strips away the abstractions to reveal the foundational mechanics of Hyperliquid's specialized L1, exposing how sub-second finality, deterministic margining, and native clearinghouse operations dictate high-frequency trading success.

2.1 Hyperliquid L1 Architecture and HyperEVM

Hyperliquid's L1 is designed around a practical division of labor: (i) a native execution environment, *HyperCore*, that runs the latency-critical exchange and risk logic, and (ii) an EVM-compatible environment, *HyperEVM*, that enables smart-contract programmability without moving liquidity or state onto a different chain. The key point for a trader or quant is that you are interacting with *one* consensus sys-

tem (HyperBFT) that secures *both* environments; HyperEVM is **not** a separate chain. This unified security model is what allows on-chain applications to be built around the same venue state that the matching and clearing logic uses, rather than wrapping, bridging, or replicating it elsewhere.

2.1.1 Two execution environments, one venue state

From a systems viewpoint, Hyperliquid's architecture is easiest to reason about if you treat HyperCore as the *engine room* and HyperEVM as the *extension bay*:

- **HyperCore:** purpose-built, non-EVM logic for the exchange itself. It is where latency-sensitive operations live: validating orders against precision rules, sequencing updates, matching trades, and updating margin/positions as part of a single coherent state transition.
- **HyperEVM:** an EVM-compatible runtime that allows smart contracts to exist *inside* the same L1 security domain. HyperEVM is aimed at programmability: building applications that can read from, and in some cases interact with, core exchange primitives (for example, integrating with the native spot and perp venues) while inheriting the same underlying consensus security.

This separation is not cosmetic. For on-chain perps, most performance and correctness constraints arise from the matching and risk pipeline, not from generalized contract execution. HyperCore can therefore be engineered narrowly around exchange throughput and deterministic portfolio accounting, while HyperEVM provides an interface for strategies and products that need composability rather than microsecond-level determinism.

2.1.2 What “not a separate chain” means in practice

Many platforms add programmability by deploying an EVM rollup or sidechain and bridging assets/liquidity back and forth. Hyperliquid’s documented framing is different: HyperEVM is *not* a separate chain and is secured by the same HyperBFT consensus that secures HyperCore. For a quant, that yields two practical implications:

- **No bridge-induced state fragmentation.** If your application needs access to the venue’s liquidity or state (positions, balances, live markets), you are not forced to mirror that state in a separate execution domain and then manage bridge latency, bridge failure modes, and inventory mismatches across domains.
- **One finality regime.** When the system commits a state transition, both the exchange-native logic and EVM-side logic sit behind the same notion of finality. That matters for automation: you can structure bots and contract flows around a single confirmation model rather than juggling “fast-but-reversible” vs “slow-but-final” regimes.

Importantly, *one security domain* does *not* mean *no separation*. HyperCore and HyperEVM remain distinct execution environments; the point is that they are secured and finalized together, which is exactly what you want when building low-latency trading systems that must reconcile state reliably.

2.1.3 Why traders should care: latency, determinism, and failure modes

Latency sensitivity lives in HyperCore. Any strategy that depends on queue position, spread capture, or fast inventory rebalancing is governed by the mechanics in the native exchange path: order acceptance, sequencing, and matching. Even if your strategy logic is authored in a contract, the outcome still depends on the HyperCore pipeline that updates fills and portfolio state.

Programmability lives in HyperEVM. HyperEVM is where you place logic that benefits from composability: vault accounting, strategy wrappers, automated execution policies, and integrations with other on-chain components. The trade-off is that generalized execution comes with different resource constraints than native matching logic; you treat it as a powerful interface layer, not a substitute for the core matching engine.

State consistency is the central design goal. For automation, the worst operational failures are the “split brain” cases: your strategy believes it has inventory when it does not, or it believes a hedge exists on another domain but it is delayed. A unified L1 with distinct execution environments is best understood as an attempt to *keep the authoritative trading state in one place* while still enabling application-layer creativity.

2.1.4 Composability: what you can assume without over-assuming

Hyperliquid’s documentation emphasizes that HyperEVM can interact directly with parts of HyperCore (including the spot and perp venues). For building systems, a clean mental model is:

- **HyperCore exposes “surfaces”** (market state, account state, and venue actions) that are meaningful to trading.
- **HyperEVM contracts can compose with those surfaces** to build higher-level behaviors (e.g., a vault that routes trades, or a strategy contract that reacts to market conditions).

Avoid assuming unverified scheduling details between the two environments. In practice, robust systems are designed to be *state-reconciled* rather than *timing-assumed*: you build so that any action that depends on a fill, balance change, or position update is gated on confirmed state rather than optimistic sequencing expectations.

2.1.5 Design takeaway for bots and on-chain strategies

Use HyperCore for what it is optimized for (execution and portfolio updates), and use HyperEVM for what it is optimized for (programmable coordination and composability), while engineering your system around one finality regime and one authoritative venue state.

Concretely, when you later implement automated execution, your fastest loop (quoting, cancel/replace, inventory constraints) should be written assuming HyperCore's rules and constraints dominate. When you implement higher-level logic (capital allocation, rule-based gating, vault shares, or strategy packaging), HyperEVM becomes the natural place to host that logic—provided you still reconcile against the venue's actual state transitions rather than treating contract execution as a guarantee of immediate market-side outcomes.

2.2 HyperCore Components and Consensus

The previous section separated HyperCore (native exchange execution) from HyperEVM (smart-contract extensibility) while emphasizing that both sit under the same HyperBFT security domain. This section stays inside HyperCore and names the primitives that matter most to trading systems: (1) how an order becomes a confirmed state update, (2) how portfolio constraints are enforced, and (3) how price inputs propagate into fills, risk checks, and forced actions.

A useful way to read HyperCore is as a tight loop that continuously transforms *intent* (orders) into *state* (positions and balances) using three coupled subsystems:

- **1. Consensus and sequencing (HyperBFT):** decides the authoritative ordering of state transitions and provides one-block finality.
- **2. Trade execution (central-limit matching):** converts orders into fills under deterministic rules.

- **3. Clearing and risk (clearinghouse + margin logic):** updates positions/balances and enforces portfolio constraints using venue and oracle-derived prices.

2.2.1 From order submission to final state: what gets decided where

At a high level, the system must answer two questions for every order:

- *Ordering:* In what exact sequence are requests applied?
- *Validity:* Given current state and risk rules, is the request admissible, and if so what state transition does it produce?

HyperBFT addresses the first question by producing an agreed-upon ordering and finalizing the resulting state update in one block. HyperCore then deterministically applies execution and clearing rules to that ordered stream. For a bot, the practical implication is that “did my cancel/replace land before someone traded through my quote?” is fundamentally an ordering question—so the relevant uncertainty window is the time until the next finalized block, not an extended probabilistic confirmation tail.

2.2.2 Consensus: why finality changes how you engineer execution

HyperBFT’s one-block finality is not merely a blockchain trivia detail; it changes the control logic of automated trading.

On probabilistic chains, a common pattern is “submit, then wait N blocks, then act,” because early confirmations can be reversed. That delay forces wider safety buffers: you quote less aggressively, you hedge later, and you keep more slack margin because state uncertainty persists.

With one-block finality, the engineering problem becomes more like a traditional venue: you still have latency, contention, and queue dynamics, but once the block is finalized you can treat the resulting portfolio

state as settled for downstream decision-making. That enables tighter feedback loops such as:

- faster reconciliation between expected vs realized position after bursts of partial fills,
- more aggressive cancel/replace policies without carrying long “pending” ambiguity,
- cleaner separation between *pre-trade* risk limits (what you allow your bot to send) and *post-trade* risk actions (how you respond to confirmed state).

2.2.3 Execution primitive: deterministic fills from a central limit book

Inside HyperCore, execution is driven by the central limit book rules you already know from off-chain perps: orders either (i) remove liquidity immediately (cross the spread) or (ii) rest and compete for queue position at a price level. The key is determinism: given the finalized ordering of actions from HyperBFT and the current book state, the set of fills is a pure function of those inputs.

For practitioners, two consequences matter:

- **1. Queue position is ultimately about ordering.** Your edge is not “being on-chain,” it is choosing when and where to place orders so that your actions land in favorable sequence relative to others.
- **2. Your backtest must model discrete events.** Continuous-time assumptions (“I can always cancel before getting hit”) break down when the system processes many actions per block. A better mental model is *batch-processed but finalized* updates.

2.2.4 Clearinghouse: turning fills into portfolio constraints

Trade execution is only half the story. The clearinghouse updates:

- positions (size, average entry, realized/unrealized PnL bookkeeping),
- balances and collateral usage,
- margin status and any constraints that gate future actions.

This is where “exchange mechanics” meet “chain mechanics”: the venue is not just matching buyers and sellers; it is simultaneously maintaining a coherent risk ledger so that every fill is immediately reflected in whether the account can continue to place orders, increase exposure, or must reduce it.

For cross-margined accounts (as discussed later in detail), this implies that *portfolio interactions are real-time*: profits in one instrument can support exposure elsewhere, and losses can tighten constraints everywhere. For isolated configurations, the same clearing logic is applied but collateral is ring-fenced at the position level.

2.2.5 Pricing inputs: oracle, mark construction, and why venue liquidity matters

HyperCore relies on external price information, but it does not treat it as a single static number.

Oracle publication is validator-driven and frequent. Validators publish oracle prices on a short cadence. Each validator computes a **weighted median over multiple CEX spot mid prices with specified weights**, and the clearinghouse aggregates validators via a **stake-weighted median**. This structure is designed to dampen the influence of outliers and make the oracle robust to single-venue anomalies.

Mark depends on both external prices and venue state. Liquidations depend on the **mark price**, and the mark price **incorporates both external prices and Hyperliquid’s own book state**. The practical consequence is subtle but important: the risk engine is sensitive to *two* liquidity conditions at once:

- dislocations in external markets (through the oracle inputs), and

- internal depth and skew (through venue state contributing to mark).

So even if the external oracle is stable, a thin internal book can still influence the mark used for forced actions; conversely, deep internal liquidity can reduce the chance that transient external prints immediately translate into forced closes.

2.2.6 Finality versus probabilistic confirmation: trading implications

You should treat finality as a property that changes which uncertainties dominate your PnL.

What finality reduces. It reduces *state reversal risk*: once your fill/cancel is finalized, you do not need to price in a long reorg tail where a confirmed trade might disappear. This is especially valuable for:

- tightly coupled hedge legs (spot/perp, perp/perp),
- inventory-sensitive quoting where you must know if you are long or flat before placing the next quote,
- systematic risk throttles based on current margin usage.

What finality does not remove. It does not eliminate:

- adverse selection (you can still be traded against when your quote is stale),
- competition for ordering within the next block,
- risk from fast mark movements driven by external prices and internal liquidity conditions.

A robust system therefore uses one-block finality to accelerate *reconciliation* (closing the gap between intended and actual state), while still treating execution as a microstructure problem (queue competition, spread dynamics, and fast price moves).

2.2.7 Closed-loop intuition: price feeds, venue state, and risk actions

HyperCore’s most important “systems” lesson is that pricing and risk form a closed loop:

- **1.** external markets influence validator oracles (via weighted medians of CEX spot mids),
- **2.** oracles and venue state jointly influence mark,
- **3.** mark influences risk checks and forced actions (including liquidations),
- **4.** forced actions and discretionary risk reductions change venue state (depth, skew, and available liquidity),
- **5.** venue state feeds back into mark construction.

For quants, this means your exposure management cannot be purely “price reactive.” You also need to be *liquidity aware*: internal depth and skew can affect the price used for risk actions, which in turn can reshape the very liquidity conditions your strategy relies on.

2.3 Instrument Specifications and Identifiers

With the execution and clearing loop in mind, the next practical problem for automation is more mundane but just as decisive: *constructing orders that the venue will accept*. Most “mysterious” rejections in early bot deployments are not alpha decay—they are formatting failures: wrong instrument identifiers, wrong precision, or price/size values that violate increment rules.

This section focuses on three things you must get right before you send any order action:

- **Instrument identity:** which market you think you are trading versus what HyperCore considers that market.

- **Metadata-driven constraints:** how you discover contract specs and per-market precision rules from venue metadata.
- **Rounding and serialization discipline:** how you map your continuous model outputs (floats) into discrete, valid order parameters.

2.3.1 Contract specification: what a “perp contract” is on Hyperliquid

Hyperliquid perpetuals are **linear perpetuals: 1 contract = 1 unit of the underlying spot asset**. If you are long q contracts of an asset with price P , your notional exposure is approximately $q \cdot P$ in quote terms (before leverage and margin effects).

A detail that affects PnL attribution and hedging is the collateral and oracle denomination mismatch: **collateral is USDC while many oracles are USDT-denominated without conversion**. Operationally, treat this as a *quanto-like* wrinkle: your margin ledger is in USDC, while key pricing inputs may be produced in USDT terms, and the protocol does not apply a USDC/USDT FX conversion. For most liquid majors this is a small basis, but for tight carry or low-volatility arbitrage it should be in your error budget.

2.3.2 Identifiers: symbols are not enough

Human interfaces show symbols; automated order routing requires *canonical identifiers*. Hyperliquid has multiple instrument “surfaces” (perps, spot, outcomes), and identifiers differ across them.

Perps: integer asset IDs.

Order-routing identifiers often require integer asset IDs. For core perps, the authoritative mapping is tied to the coin index in the perps universe returned by metadata (commonly referred to as the meta response in docs). A frequent mistake is to hardcode “BTC = 0” from a particular environment and assume it will hold everywhere; for robust systems, you resolve IDs dynamically from metadata on startup and periodically refresh.

Builder-deployed perps (HIP-3): encoded indices and names.

Builder-deployed perps use an encoded asset scheme of the form

$$\text{asset} = 100000 + \text{perp\_dex\_index} \cdot 10000 + \text{index\_in\_meta},$$

and are named $\{\text{dex}\}:\{\text{coin}\}$. The key implication is that “coin name” alone is insufficient to disambiguate markets; your bot needs to carry both the canonical name and the correct integer identifier.

Spot: special coin naming and separate IDs.

Spot uses special coin identifiers. Two conventions matter operationally:

- PURR/USDC is explicitly used for PURR spot.
- Otherwise, spot instruments commonly use the @index convention derived from the spot universe in spotMeta.

In addition, **spot asset IDs are** $10000 + \text{spotInfo.index}$ **from** spotMeta. Critically, spot IDs differ from token IDs, and **they can differ between mainnet and testnet**. If you trade both environments, treat the “ID layer” as environment-specific configuration, not shared code constants.

UI symbol remapping is a real pitfall.

Some user interfaces remap symbols (for example, showing a familiar BTC/USDC label that corresponds to a different canonical coin name internally). For a bot, always use canonical identifiers returned by the metadata endpoints rather than screen labels or informal tickers.

2.3.3 A bot-grade resolution workflow (metadata first, orders second)

A robust order-construction pipeline is deterministic and metadata-driven:

- **Resolve the target market** from canonical metadata: fetch the universe (meta for perps, spotMeta for spot), and select the exact instrument by its canonical coin/name fields.

- **Extract routing identifiers:** compute or read the integer asset ID required by order actions, or the spot coin identifier where applicable.
- **Extract precision constraints:** read per-market fields such as `szDecimals` and any other constraints needed to form valid price and size values.
- **Quantize your model outputs:** map continuous target price/-size to the discrete grid imposed by tick/lot/decimal rules.
- **Serialize conservatively:** avoid binary float surprises by formatting to a bounded decimal representation consistent with venue rules.

Even if you cache metadata, design your bot to handle changes: new listings, parameter updates, or differences across environments. In production, it is common to refresh metadata on a schedule and to treat any “invalid precision” rejection as a signal to re-pull specs.

2.3.4 Precision constraints: sizes are decimal-quantized, prices are precision-limited

Hyperliquid validates both size and price formatting. Two constraints from the docs are particularly important:

Size uses `szDecimals`-based rounding.

Sizes must be rounded to the asset’s `szDecimals`. If the venue says `szDecimals = 3`, then valid sizes are multiples of 10^{-3} . Any size with higher precision than allowed is rejected.

A safe implementation is:

$$size_{valid} = \lfloor size \cdot 10^d \rfloor / 10^d, \quad d = szDecimals,$$

with the caveat that rounding *direction* is a strategy choice:

- For *opening* orders, rounding down (flooring) reduces the chance of exceeding intended risk limits.

- For *closing* orders, rounding up may reduce residual dust positions, but can introduce brief over-close behavior depending on current position and venue semantics.

In practice, many bots implement side-aware quantization: floor for buys when constrained by margin, and floor for sells when constrained by inventory, while using “close-only” style logic where supported (if not supported, you emulate it by capping size to current position).

Price has significant-figure and decimal-place limits.

Tick rules are not only “0.10 increments” style. Hyperliquid imposes **significant-figure constraints for price** and also a **max decimal-place rule based on** `MAX_DECIMALS - szDecimals` (with `MAX_DECIMALS` differing between perps and spot). The practical takeaway is that you cannot assume “any number of decimals is fine as long as it looks like a tick”; the venue can reject a price that has too many decimals or violates significant-figure limits even if it is close to a plausible level.

Because `MAX_DECIMALS` is venue-surface-specific and the exact limit is enforced server-side, the safest approach is:

- format prices using decimal arithmetic (or string formatting) rather than raw IEEE-754 floats,
- clamp decimal places to the maximum allowed for that market,
- optionally enforce a significant-figure cap in your formatter if you are generating very small prices (low-priced tokens) where significant-figure rules tend to bite.

2.3.5 Why these rules exist: discrete books and deterministic risk

These constraints are not arbitrary bureaucracy; they serve two venue-level requirements:

Discrete state space.

Order books are stored on a discrete grid. If arbitrary precision were

allowed, two orders that are “economically identical” could occupy different price levels, fragmenting liquidity and complicating matching.

Deterministic accounting.

Margining and clearing must be reproducible and robust. Tight bounds on decimals reduce ambiguity in serialization and prevent edge cases where a client and server interpret “the same” float differently.

2.3.6 Operational failure modes and how to eliminate them

Below are the most common implementation failures and the corresponding fixes:

- **Trading the wrong instrument due to symbol confusion.** Fix: only trade instruments selected from `meta/spotMeta`; never map UI symbols manually.
- **Wrong asset ID in multi-environment deployments.** Fix: resolve IDs at runtime per environment; do not reuse mainnet constants on testnet.
- **Precision rejections from float noise.** Fix: quantize using integer math on scaled values or decimal types; serialize with bounded decimals.
- **Inconsistent rounding between signal and execution.** Fix: incorporate quantization into your signal pipeline so your model learns the true discrete action space (especially important for tight spreads and small sizes).

2.3.7 Minimal checklist for “order validity” in automated execution

Before any order action leaves your system, it should pass a local validity gate:

- Market resolved from fresh metadata (`meta / spotMeta`)

- Correct routing identifier (asset integer for perps; correct coin naming for spot, including PURR/USDC and @index conventions)
- Size quantized to `szDecimals`
- Price formatted to satisfy both decimal-place and significant-figure constraints (surface-specific; do not assume perps and spot share limits)
- Environment-aware ID derivations (spot ID 10000 + `spotInfo.index`; builder-perp encoding when applicable)

Treat this gate as part of your core execution stack, not a convenience. When you later optimize quoting and cancellation rates, the fastest bot still loses if a nontrivial fraction of its messages are rejected for avoidable formatting reasons.

2.4 Margining and the Perpetual Risk Engine

Once you can generate valid orders, the next constraint is whether your *portfolio* is allowed to hold the resulting exposure. HyperCore’s risk engine is deliberately mechanical: it evaluates collateral, unrealized PnL, and position notionals under explicit formulas, and it enforces forced de-risking when equity falls too far. For systematic traders, this is not just “safety plumbing”—it is a first-order input into position sizing, inventory limits, and the maximum turnover you can run without getting clipped during volatility spikes.

This section focuses on the algorithmic parts you must model:

- the two margin modes (cross vs isolated) and what they imply for portfolio coupling,
- the relationship between leverage, initial margin, and maintenance margin,
- tiered maintenance requirements for large notionals,
- liquidation and the backstop path via the liquidator vault.

2.4.1 Cross margin vs isolated margin: coupling versus containment

Hyperliquid supports two margin regimes:

- **Cross margin: collateral is shared across positions.** Gains and losses from one position affect the free collateral available to all others. This increases capital efficiency (one pool of collateral supports multiple exposures) but couples your risk: a bad move in one instrument can impair unrelated positions.
- **Isolated margin: collateral is ring-fenced by asset/position.** The intent is containment: a blow-up in one isolated position should not automatically drag the rest of the account into distress.

Some assets may be **isolated-only with restrictions on margin removal**. From a systems perspective, this is the venue's way of constraining tail behavior in instruments that are more prone to gaps or thin liquidity.

A trading desk analogy is helpful: cross margin resembles a single firm-wide balance sheet; isolated resembles separate accounts per desk. Cross is efficient but contagious; isolated is safer but leaves idle collateral "stuck" behind walls.

2.4.2 Initial margin: the admission ticket for new exposure

Hyperliquid's initial margin requirement for opening exposure is explicit.

Initial margin to open is

$$\frac{\text{position\_size} \cdot \text{mark\_price}}{\text{leverage}}.$$

Two operational points matter:

Leverage is checked at open. Leverage checks occur at open, meaning the venue decides whether you are allowed to create or increase exposure based on current collateral and the requested leverage.

After that, **ongoing leverage management is the trader responsibility**: if the position moves against you, your effective leverage rises as equity falls, and the venue will not “auto-delever” you gently—it will enforce maintenance rules and then liquidate if needed.

Your sizing model should use the same price input the venue uses. The formula references mark, not last trade. As discussed earlier, the venue’s pricing input is designed to be harder to manipulate and more conservative under stress. When you compute “how much can I buy?”, using last price instead of mark can systematically overestimate capacity during fast moves.

2.4.3 Maintenance margin: the survival threshold

Initial margin is about opening; maintenance margin is about *staying open*. Hyperliquid’s maintenance requirement is not necessarily a single flat percentage for all sizes. Instead:

- **Maintenance margin can be tiered by notional size.**
- **The maintenance margin function is constructed to be continuous across tiers.**
- **Tier configs are referenced by IDs discoverable via `meta`.**

The continuity property matters for large traders: it prevents pathological “cliffs” where adding \$1 of notional suddenly jumps maintenance by a large discrete amount. But continuity does *not* mean smoothness is irrelevant—near tier boundaries, your effective buffer can change meaningfully, and a position that is safe at one size may become fragile after a small scale-up.

Large-notional intuition. As notional grows, venues typically increase maintenance requirements to reflect gap risk and the difficulty of liquidating size without moving the market. Tiering is the mechanism that converts that intuition into deterministic rules your bot can anticipate.

2.4.4 Equity decay, gap risk, and liquidation triggers

Liquidations are the venue's last resort to prevent negative equity. The key facts for Hyperliquid's mechanism are:

- **Liquidation uses mark price.**
- If normal liquidation through the book fails, there is a **backstop liquidation path through a liquidator vault.**
- **Cross vs isolated backstop outcomes differ materially.**

"Gap risk" is the reason liquidation logic cannot be modeled as a gentle, continuous process. If price moves sharply between updates, equity can fall through the maintenance threshold faster than the position can be reduced. The venue therefore needs a deterministic procedure to transfer risk out of a distressed account even when in-venue liquidity is insufficient.

2.4.5 A simple equity-versus-threshold picture

A clean way to reason about liquidation conditions is to track *equity* over time against two thresholds:

- **Initial margin** (entry constraint): above this, you can open/increase positions at the chosen leverage.
- **Maintenance margin** (survival constraint): below this, you are eligible for forced reduction and liquidation procedures.

The chart below uses a stylized equity path to convey the logic: equity decays as the position moves against you and/or as volatility increases requirements; once equity falls below maintenance, the account enters a forced-deleveraging regime, and if equity continues to erode the position can be liquidated.

2.4.6 Backstop liquidation and why cross vs isolated outcomes differ

When liquidation can be executed through the venue (i.e., sufficient liquidity exists to close or reduce the distressed position), the result resembles a forced market unwind at unfavorable prices. The more important tail case is when that is not feasible fast enough. Hyperliquid includes a **backstop liquidation path through a liquidator vault** for these situations.

The distinction between cross and isolated is economically large:

Cross margin backstop can transfer the whole cross portfolio. Because cross positions share collateral, the system may need to treat the portfolio as a coupled object. In extreme stress, the backstop mechanism can effectively “take over” the cross account’s exposures and associated margin, since leaving some positions behind can preserve hidden negative-equity pathways.

Isolated margin tends to localize the damage. With ring-fenced collateral, backstop outcomes can be constrained to the isolated position and its margin allocation. That makes isolated mode attractive for highly convex, gap-prone exposures where you want an upper bound on portfolio contamination.

For systematic traders, the lesson is that margin mode is not a cosmetic UI toggle; it is a structural choice about whether you accept portfolio-level contagion in exchange for capital efficiency.

2.4.7 Modeling guidance: what to include in sizing and execution logic

To incorporate these mechanics into a quantitative system, include the following components explicitly:

- **Pre-trade margin simulation.** Before sending an order, estimate post-trade initial margin usage using

$$\frac{\Delta \text{ position\_size} \cdot \text{mark}}{\text{leverage}}$$

and include a buffer for mark slippage during fast markets.

- **Tier-aware maintenance.** Pull tier configuration IDs (via meta) and compute maintenance requirements as a function of notional so your strategy does not accidentally step into a tighter regime.
- **Mode-aware drawdown limits.** In cross mode, enforce portfolio-level limits (max aggregate notional, max correlated exposure). In isolated mode, enforce per-position loss limits that assume the rest of the account remains intact.
- **Stress testing for gaps.** Simulate instantaneous mark moves that jump across maintenance thresholds without intermediate trading opportunities; this is where many “it looked safe” leverage levels fail in live trading.

The common failure pattern is optimizing a strategy on smooth historical paths and then discovering that the real constraint is not average volatility—it is the discrete event of a fast move that pushes equity under maintenance, triggers forced actions, and reshapes your inventory at the worst time. Modeling the margin engine with the venue’s own primitives is how you prevent that class of blow-up.

2.5 Funding Mechanics and Strategy Implications

Perpetuals remove expiry, but they do not remove carry. Funding is the mechanism that continually nudges the perpetual price toward the underlying reference by transferring cash between longs and shorts. As we saw in the pricing and risk loop, HyperCore draws on oracle inputs and venue liquidity to construct the prices that drive accounting. Funding sits alongside that loop: it is a *separate* cashflow stream from mark-to-market PnL, and it is paid on a deterministic schedule with its own notional convention.

This section focuses on three practical questions:

- What exactly is the funding rate and how is it produced?
- When and on what notional is funding transferred?

- How should a quant decompose and forecast PnL when funding is a major driver?

2.5.1 Peer-to-peer transfers on an hourly clock

Hyperliquid funding is **peer-to-peer and paid hourly**; there is no protocol fee collected from the funding transfer itself. The protocol expresses the funding rate using an **8-hour convention** and then applies it **as one-eighth per hour**. In other words, if the displayed (or reported) funding rate corresponds to an 8-hour rate f_{8h} , then the hourly transfer rate applied at each settlement is approximately:

$$f_{1h} = \frac{f_{8h}}{8}.$$

This schedule choice matters for execution and risk:

- Funding accrues and is exchanged frequently, which reduces the time window over which mispricings can persist without cash-flow consequences.
- “Funding capture” strategies are sensitive to hour boundaries. Entering a position just before settlement is economically different from entering just after, even if prices are unchanged.

2.5.2 Where the rate comes from: interest + premium, premium from impact vs oracle

The funding rate combines an **interest component convention plus premium component**. The premium is the economically important part in active markets: it encodes whether the perpetual is trading rich or cheap relative to the underlying reference.

Hyperliquid’s premium construction is tied to **impact prices** and the oracle:

- The **premium derives from impact prices vs oracle price**.
- **Premium sampling is high-frequency and averaged**.

Two intuitions follow:

Impact price makes funding liquidity-aware. Using impact prices (rather than, say, last trade) makes the premium sensitive to how much you would move the market to trade a meaningful size. When the venue is thin or skewed, impact prices can diverge from the mid, and the premium reflects that. For traders, this ties funding directly to order-book conditions: funding can become expensive precisely when the book is one-sided.

High-frequency sampling reduces “single print” gaming. Averaging premium samples over time makes it harder for one transient move to dominate the funding rate. It also means your model should treat the funding rate as the outcome of an averaging process rather than a point-in-time snapshot.

2.5.3 Notional convention: funding uses oracle price, not the venue’s mark

A crucial accounting detail is the notional used for funding transfers.

Funding payments use oracle price (not mark price) to compute notional.

This creates a deliberate separation of roles:

- The venue’s mark is used for risk and forced actions (liquidation and related mechanics).
- The oracle price anchors the funding notional for the periodic cashflow exchange.

For hedgers, this matters because a “perfect hedge” depends on which price you are hedging:

- Your unrealized PnL evolves with the venue’s valuation conventions (mark-to-market),
- while your funding cashflows scale with oracle notional.

When oracle and mark diverge during stress (they can, because mark incorporates venue state), you can see short intervals where hedge ra-

tios that look correct under one convention leave residual exposure under the other. The impact is usually second-order for liquid majors but can be first-order in thin markets or during fast dislocations.

2.5.4 Funding caps: bounding extreme carry

Hyperliquid documentation notes that **funding caps exist and apply across assets**. Conceptually, caps limit the magnitude of transfers per unit time, preventing runaway carry in extreme one-sided conditions. For strategy design, treat caps as:

- a bound on worst-case funding bleed when you are stuck on the wrong side of a crowded trade,
- and a bound on best-case “carry harvest” when you are on the right side.

If you backtest a carry strategy in regimes of extreme premiums, failing to include caps can materially overstate returns.

2.5.5 PnL decomposition: separate price PnL from funding cashflows

A clean quantitative decomposition is:

$$\text{Total PnL} \approx \underbrace{\Delta \text{Position Value}}_{\text{mark-to-market}} + \underbrace{\text{Funding Cashflows}}_{\text{hourly transfers}} - \underbrace{\text{Fees}}_{\text{execution costs}}.$$

Funding is not “free yield”; it is a transfer that compensates the side that is structurally underpaid in price terms. For modeling and attribution, keep two ledgers.

1) Mark-to-market ledger. This is your PnL from price movement under the venue’s valuation rules (what your position is worth now versus before). This is what dominates in trends and volatility spikes.

2) Funding ledger. This is the deterministic sequence of hourly transfers computed from:

- your position size (contracts),
- the *oracle* notional,
- and the hourly funding rate f_{1h} .

In discrete time, a simplified funding cashflow approximation for a constant position q over one hour is:

$$\text{Funding}_{t \rightarrow t+1h} \approx -q \cdot P_t^{\text{oracle}} \cdot f_{1h,t},$$

with the sign determined by whether longs pay shorts or vice versa.

This separation is essential for diagnosing “mysterious drift” in live PnL: a strategy can be directionally correct but still lose due to persistent negative funding, or it can be directionally flat but profitable because it consistently earns carry.

2.5.6 Strategy implications

1) Hourly settlement turns timing into edge (or slippage)

Because funding is paid hourly, entry and exit timing relative to the hour boundary changes realized carry. A carry-oriented strategy should:

- explicitly model the next funding timestamp,
- estimate whether the expected funding over the holding window exceeds fees and expected adverse selection,
- avoid “accidental funding” (opening exposure that is expected to be held through settlement when the rate is unfavorable).

2) Premium is liquidity-aware, so crowding risk appears as funding risk

Since the premium is built from impact prices versus oracle, funding can become punitive when positioning is one-sided and liquidity thins. This is not just a cost; it is a *signal* of crowding and potential instability:

- Positive funding (longs paying) often coincides with levered long crowding; reversals can be violent.

- Negative funding (shorts paying) often coincides with crowded shorts; squeezes can be sharp.

A robust system treats extreme funding as a risk state: reduce leverage, widen quoting bands, and cap inventory accumulation.

3) Basis trades must hedge both price movement and funding conventions

A common structure is a “delta-neutral” basis trade (e.g., long spot/short perp or vice versa). On Hyperliquid, you must align:

- the hedge ratio (contracts vs spot units),
- the fact that funding notional uses oracle price,
- and the possibility that mark and oracle diverge transiently.

The practical result is that even a neutral basis book can experience short-term equity swings due to valuation convention differences, especially in thin conditions. Your margin buffer should be sized accordingly, particularly if the basis trade is levered.

2.5.7 Implementation note for bots: model funding as a scheduled cashflow

In automated execution, treat funding like a deterministic “dividend schedule” with an hourly cadence:

- Forecast expected funding over your intended holding horizon using the premium process (impact-vs-oracle) rather than a naive last observed rate.
- Attribute realized PnL with two components (price PnL and funding) so that optimization does not confuse carry with alpha.
- Incorporate caps into both backtests and live risk limits to avoid overestimating carry in extremes.

The end result is a more accurate expected value calculation:

$$\mathbb{E}[\text{Net PnL}] = \mathbb{E}[\Delta \text{Position Value}] + \mathbb{E}[\text{Funding}] - \mathbb{E}[\text{Fees}],$$

where funding is not an afterthought—it is often the dominant term for mean-reversion, inventory-heavy, and basis-oriented approaches.

2.6 Fee Structures and Rebates

After instrument formatting, margin constraints, and funding cash-flows, the next determinant of realized performance is fee economics. For high-turnover strategies, fees are not a rounding error; they are often the dominant term in the expectancy equation. The reason is simple: in a tight-spread venue, your gross edge per trade is measured in basis points, while your costs are also measured in basis points and apply mechanically to every fill.

This section focuses on how Hyperliquid’s fee schedule and rebates influence turnover-heavy approaches, and how to incorporate them correctly in backtests and live PnL.

2.6.1 The fee state is account-specific and time-varying

Hyperliquid fees are not a single static number. The key operational facts are:

- **Fees are based on rolling 14-day volume, assessed daily (UTC).**
- **Sub-accounts share tiers with master accounts.**
- **Spot volume is double-weighted for tiering.**

Treat this as a *state variable* of your trading system. Your fee tier today is a function of what you traded over the last 14 days, and it updates on a daily schedule. For quants, the implication is that a strategy’s economics can drift over time even if its microstructure edge is unchanged, simply because your tier changes.

Two practical consequences follow:

(1) Backtests must model tier evolution or at least bracket it.

If you backtest with a fixed “best tier” but deploy at a worse tier, you will systematically overestimate net profitability. Conversely, if your strategy’s turnover pushes you into better tiers, your live results can improve after a ramp-up period.

(2) Sub-account structure affects optimization.

Because sub-accounts share tiers with the master, splitting strategies across sub-accounts does not reset you to a worse tier. That is useful operationally (separating inventory and risk), but it also means you should model fee tiers at the master-account level when allocating strategies.

2.6.2 Maker rebates and taker fees: the sign matters

On many venues, makers pay less than takers. On Hyperliquid, the maker side can receive rebates:

- **Maker rebates are paid continuously per trade to the trading wallet.**

This single sentence has a large modeling implication: for liquidity-supplying strategies, the fee line item can be *negative* (a credit), not merely “less positive.” That changes how you think about break-even spread capture. A strategy can be viable even when the raw spread capture is thin, provided it earns sufficient rebate and keeps toxic flow under control.

There is an additional detail that affects long-run tier outcomes and instrument selection:

- **Aligned quote assets get better fee/rebate terms and extra tier-volume credit.**

For an automated strategy, this means the “best” market is not only the one with the cleanest signals; it may be the one whose fee/rebate terms and tier credit accelerate you into better economics.

2.6.3 HIP-3 markets: fee economics may differ by market family

Hyperliquid includes builder-deployed markets (HIP-3) whose economics can diverge from core listings:

- **HIP-3 markets can have distinct economics (fee splits with deployers; growth mode reduces protocol-side accounting by 90% per official fees doc; community wiki provides additional detail).**

From a trader's perspective, you should not assume that a fee tier implies the same net costs across all markets. For a multi-market bot, treat "market family" (core vs HIP-3) as an input into expected net fees and rebates, and validate the schedule per instrument rather than assuming uniformity.

2.6.4 Turnover math: expected value with fee and rebate terms

To keep the logic clean, write a per-fill expected value decomposition in quote terms (e.g., USDC):

$$E[\text{Net}] \approx E[\text{Gross edge}] + E[\text{maker rebate}] - E[\text{taker fee}] - E[\text{microstructure loss}].$$

Here each term corresponds to: $E[\text{Gross edge}]$ = spread capture / signal; $E[\text{maker rebate}]$ (≤ 0 cost (credit)); $E[\text{taker fee}]$ (≥ 0 cost); $E[\text{microstructure loss}]$ = toxic flow, slippage.

For a liquidity-supplying strategy, you typically have a mix of fills:

- **Passive fills** that earn the maker rebate (or lower fee), and
- **Active hedges** that pay taker fees when you need to rebalance inventory quickly.

The economic question is therefore not "maker vs taker" in isolation; it is the *ratio* of passive to active trading required to control inventory.

If your inventory control requires frequent active trades, taker fees can dominate your edge even if your quoting looks profitable on paper.

2.6.5 Why execution constraints feed back into fee economics

Fee modeling is incomplete if it ignores operational limits. Hyperliquid’s documentation highlights that API/user limits exist (rate limits, connection/subscription limits, and degraded behavior under throttling). The key trading implication is:

If congestion throttles reduce your ability to maintain passive quotes (cancel/replace, widen/narrow with volatility, refresh on fills), you may lose maker share, trade more actively to rebalance, and therefore shift your realized fee mix in the wrong direction.

This is a coupling between infrastructure and economics: being throttled is not just “slower,” it can convert a rebate-earning flow into a fee-paying flow by forcing more emergency hedging.

2.6.6 A simple break-even framework for high-turnover liquidity provision

A convenient way to think about break-even is in “per-trade edge required” as a function of how frequently you trade. Suppose your strategy produces a small gross edge per round-trip, but you do it many times per day. Then:

- With **no maker rebate**, the cumulative fees create a fixed headwind proportional to the number of fills.
- With **maker rebates**, the headwind is reduced and can even turn into a tailwind if most fills are passive.

The chart below plots a stylized daily profit curve as trade count increases under three accounting treatments:

- gross PnL ignoring fees,

- net PnL with maker rebate (mostly passive fills),
- net PnL without maker rebate (or with worse fee terms / more active hedging).

The point is not the specific parameters; it is the shape: once turnover is high, small differences in per-trade fees or rebates compound into large differences in daily expectancy.

2.6.7 Implementation checklist: fee-aware trading and backtesting

To incorporate fee and rebate mechanics correctly, your execution stack should treat fee economics as a live input:

- **Track your current tier** (rolling 14-day volume, assessed daily UTC) and update expected costs when the day rolls.
- **Attribute fills by liquidity side** (passive vs active) so realized rebates/fees match your model's assumptions.
- **Model spot volume double-weighting for tiering** if your strategy mixes spot and perps and relies on moving tiers.
- **Include infrastructure constraints** (rate limits and throttling regimes) in simulation if your edge depends on maintaining high passive fill share.
- **Differentiate market families** (core vs HIP-3) when forecasting net costs, since economics can differ via fee splits and growth mode accounting.

The core lesson is that turnover strategies live or die on basis-point arithmetic. When you model fees, rebates, and their interaction with execution constraints, you convert what looks like “small” plumbing into a precise estimate of whether the strategy has positive expectancy after all mechanical transfers are paid.

Chapter 3

Order Execution and Microstructure Tactics

In the hyper-competitive arena of on-chain perpetuals, generating a profitable trading signal is only half the battle; executing it without leaking alpha is where true quants separate themselves. This chapter descends into the microstructure of Hyperliquid's L1 matching engine. You will learn how to navigate price-time priority, deploy surgical order flags to manage risk, and architect state-aware execution loops that turn theoretical edge into realized PnL while minimizing adverse selection and slippage.

3.1 Matching Priority and Fill Dynamics

Hyperliquid perps trade on an L1-hosted central limit order book (CLOB). As you saw earlier, the rule that decides *who gets filled first* is price-time priority: better prices trade first; for orders at the same price, earlier arrivals trade first. This section goes beyond the definition and focuses on what a quant or bot engineer actually needs: (i) how queue position translates into *fill probability and fill timing*, (ii) why passive fills are *path-dependent* (partial fills, cancellations ahead of you, and book re-pricing), and (iii) how to reason about and mitigate

the core risk of passive liquidity provision: trading just before the price moves against you.

3.1.1 Price–time priority as a queueing problem

Consider a single price level on one side of the book (e.g., the best bid). At that exact price, resting limit orders form a FIFO queue. Your order joins at the back; incoming marketable flow (market orders or aggressive limits that cross the spread) consumes the queue from the front.

For a fixed price level p , define:

- Q_{ahead} : total resting size *ahead of you* at price p at the moment your order becomes active (after any immediate matching checks, post-only logic, etc.).
- q : your order size at p .
- M_t : cumulative *marketable* volume that arrives and matches against price p on that side over horizon t (i.e., buy marketable volume for the ask queue, sell marketable volume for the bid queue).
- C_t : cumulative volume *cancelled* or re-priced *ahead of you* at that same price level over horizon t . (Cancellations behind you do not help you.)

First fill occurs when

$$M_t + C_t \geq Q_{ahead}.$$

Your *filled size* by time t is then

$$\text{Fill}_t = \min(q, \max(0, M_t + C_t - Q_{ahead})).$$

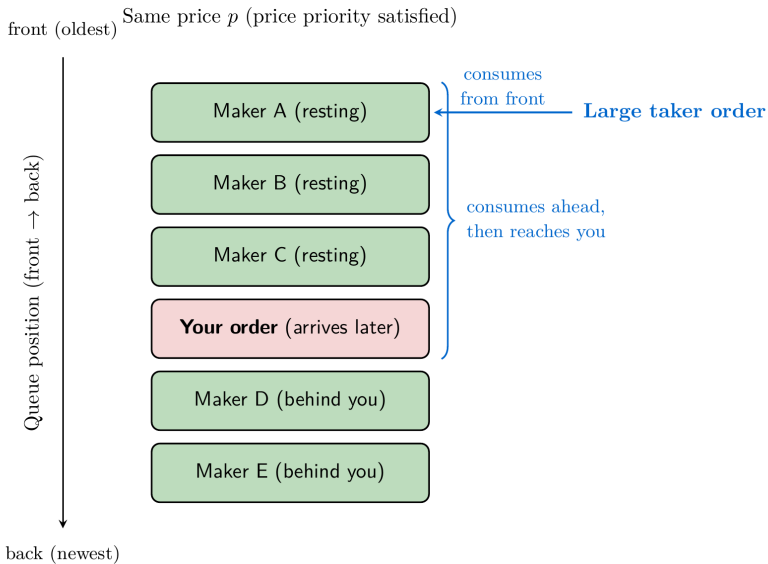
Two immediate implications matter for bot design:

- **Fill probability is not just about trade volume.** Cancellations ahead of you can be as important as taker volume. In quiet markets, a meaningful share of your progress can come from other makers pulling orders rather than from takers consuming them.

- **Queue position is a state variable.** Even if the best price does not change, Q_{ahead} evolves as other participants cancel/replace, and as partial fills occur.

The practical job of an execution system on Hyperliquid is to maintain a good enough estimate of (Q_{ahead}, M_t, C_t) to decide when to stay passive, when to reposition, and when to turn aggressive.

Queue at a single price level (FIFO within price)



Adverse Selection: If the taker flow is information-driven, passive fills can occur just before a price move, filling your order precisely when you might prefer to cancel.

3.1.2 Partial fills and the “fill path”

A common misconception is that a resting order either fills or does not. In practice, fill outcomes are *continuous* because taker volume arrives in chunks. If a marketable order consumes only part of the level, every maker at the front can experience partial execution. This matters for

three reasons:

- **Inventory arrives in pieces.** Your realized entry price and your risk exposure evolve trade-by-trade, not in one instantaneous step.
- **Cancel/replace decisions become timing-sensitive.** If you cancel to reprice, you may forfeit an almost-complete queue position; if you wait, you risk being the next liquidity to trade against informed flow.
- **Your PnL distribution changes shape.** The variance of outcomes increases because small partial fills create residual inventory that can be awkward to hedge, especially when the spread widens.

A robust mental model is to treat a resting order as a *process* with a state variable $x_t \in [0, q]$ representing cumulative filled size. Execution then becomes a sequence of micro-events:

$$x_{t+} = x_t + \Delta x_t, \quad 0 \leq \Delta x_t \leq q - x_t,$$

where Δx_t depends on (i) how much marketable volume arrives, and (ii) how much queue is still in front of you at that instant.

From an engineering standpoint, this implies your order-management loop should not be keyed to “filled/unfilled” booleans. It should be keyed to *deltas* in filled size and to the evolving estimate of queue in front, because those are the variables that determine the next action (hold, cancel, reprice, or flip to taker).

3.1.3 Maker vs taker fills: knowing what you actually did

Hyperliquid fill streams expose whether a fill was maker-side or taker-side; the gateway stream includes a boolean like `crossed` where `crossed=true` indicates you crossed the spread (taker) and `crossed=false` indicates you provided liquidity (maker). In practice, you want to persist this field per fill because it lets you:

- audit fee outcomes (maker rebates versus taker fees, where applicable),
- compute separate performance metrics (maker edge versus taker urgency cost),
- and detect unintended behavior (e.g., a strategy that was meant to be passive but is frequently taking).

This classification also clarifies the difference between *queue* risk and *impact* risk:

- As a maker, you accept queue uncertainty and selection risk in exchange for avoiding immediate impact and potentially earning rebates.
- As a taker, you pay the spread/impact deterministically but avoid waiting in line.

3.1.4 A practical fill-probability model (and what it leaves out)

You rarely need a perfect model of the full LOB to make better decisions; a *calibrated, local* approximation already improves execution. A simple starting point is to model marketable arrivals at your price as a Poisson process with rate λ (units: size per second) and to treat cancellations ahead as another independent Poisson-like process with rate μ . Under that simplification, the time-to-first-fill is approximately the first time $M_t + C_t$ exceeds Q_{ahead} .

If M_t and C_t are modeled as independent compound Poisson processes with expected rates λ and μ in volume units, then the expected time to reach the front is roughly

$$\mathbb{E}[T_{front}] \approx \frac{Q_{ahead}}{\lambda + \mu}.$$

This is not a universal law; it is a benchmark that forces your system to quantify the obvious:

- If you join a long queue (Q_{ahead} large), you should expect slow fills unless volume is high or makers are canceling aggressively.
- In a low-activity regime ($\lambda + \mu$ small), small price improvements can beat patience because they move you to a different (often shorter) queue.

What this model leaves out (and what you should add once the basics work):

- **Regime dependence.** λ and μ change with volatility, news, and funding-driven flow.
- **Endogeneity.** Your own order changes the book and can alter the flow you attract.
- **Toxicity.** Marketable volume is not always “uninformed.” Informed takers create selection risk that correlates with price moves.

3.1.5 Adverse selection: why passive fills can be costly even with rebates

Passive execution is attractive because you can earn the spread and (often) maker rebates. The catch is that the taker who hits you is frequently doing so for a reason. A useful decomposition for the expected value of a passive fill is:

$$\mathbb{E}[\text{edge}] \approx \frac{s}{2} + \mathbb{E}[\text{rebate} - \text{fees}] - \mathbb{E}[\Delta m] - \mathbb{E}[\text{inventory risk over holding period}].$$

Here:

- s is the bid–ask spread (in price units).
- Δm is the change in the midprice after your fill over a short horizon (e.g., 1–5 seconds, or the time it takes your bot to hedge/adjust).

Selection risk is the term $\mathbb{E}[\Delta m]$ turning negative for makers: you get filled, and then the price tends to move against your newly acquired position. This effect is strongest when the incoming marketable flow is information-driven (breakouts, liquidations, rapid repricing).

A very actionable metric is the post-fill drift. Define

$$D(\tau)$$

as the expected post-fill drift conditioned on a maker fill; concretely, $D(\tau)$ is the expected signed midprice change over horizon τ after a maker fill. In practice compute $D(\tau)$ by conditioning on maker fills and measuring

$$D(\tau) \approx \mathbb{E}[\textit{side} \cdot (m_{t+\tau} - m_t) \mid \text{maker fills at } t],$$

where $\textit{side} = +1$ for buys and $\textit{side} = -1$ for sells, and m_t is the mid at time t . For a healthy maker strategy, you want $D(\tau)$ to be close to zero or positive (price drifting in your favor after fills). If it is strongly negative at short horizons, you are being “picked off.”

3.1.6 Queue-position tactics that do not accidentally increase toxicity

Queue optimization is not “always be first.” It is “be first when the flow is safe enough to earn spread; be absent when the flow is toxic.” The following tactics are common, but they should be applied with explicit guardrails.

3.1.7 Join vs improve: when one tick matters

Suppose you want to buy. You can:

- **Join** the best bid: minimal price concession, but potentially large Q_{ahead} .
- **Improve** the bid (quote inside the spread, if possible): you become best bid and likely face a much shorter queue, but you give up price (and may tighten the spread for others).

A simple decision inequality compares expected wait cost (including selection risk while waiting) versus the cost of price improvement. Let δ be your price improvement (in dollars) relative to joining, and let h be your expected time-to-fill if you join. If the short-horizon adverse drift rate is approximately k dollars per second while you wait (estimated from data in similar regimes), then improving can be justified when

$$\delta \lesssim k h.$$

This is deliberately crude, but it produces a measurable rule: long queues plus fast-moving markets make improving more attractive; short queues or stable markets favor joining.

3.1.8 Cancel/replace without “thrashing” your queue

Frequent cancel/replace loops can destroy your time priority. If you repeatedly cancel and re-enter at the same price, you continuously reset to the back. That makes you look active but accomplishes little.

A practical approach is to separate *quote refresh* from *queue churn*:

- Refresh only when a state variable crosses a threshold: spread changes, mid moves beyond tolerance, toxicity metric spikes, or your inventory deviates.
- When refreshing, prefer *price changes* (move to a different queue) rather than cancel/re-enter at the same price.
- Maintain a minimum rest time unless risk constraints require immediate action.

This behavior is not about etiquette; it is about measurable execution. Thrashing increases message load, increases operational risk (race conditions between cancel acknowledgements and fills), and often worsens fill probability because you are perpetually late in the FIFO line.

3.1.9 Do not “optimize queue” by quoting through unstable microstructure

In fast moves, the best price level can be a trap: you get filled exactly when the market is repricing. Two filters help:

- **Spread/volatility filter.** When spread widens or short-horizon volatility jumps, reduce passive size or widen your quotes. The economic reason: the half-spread term is not enough to pay for the larger $\mathbb{E}[\Delta m]$ term.
- **Order-flow imbalance filter.** If aggressive flow is one-sided (e.g., repeated buy market orders lifting the ask), your resting sell liquidity is more likely to be hit by informed buyers (bad for you).

You can implement these using only top-of-book and trade prints; you do not need full depth to detect “market is running.”

3.1.10 Quantifying your queue and toxicity from streams

Even without a perfect LOB reconstruction, you can build useful estimates from the data you already consume for a bot:

- **Queue ahead estimate** Q_{ahead} . Track the size resting at your price when you place the order, then decrement it by (i) observed trades at that price that consume that side, and (ii) observed cancellations at that price if you subscribe to order/book update streams. Treat missed updates as uncertainty (widen your confidence interval).
- **Fill-rate estimate** λ . Compute rolling executed volume at your level (or at best price) divided by time. Use an exponentially weighted moving average (EWMA) so the estimate adapts to regime shifts.
- **Toxicity proxy.** Maintain a short-horizon post-trade midprice drift conditioned on maker fills, $D(\tau)$, and/or a measure of trade sign imbalance over the last N seconds.

When you later evaluate strategy performance, separate PnL into (i) maker fills versus taker fills (using `crossed`), and (ii) immediate mark-out after fills. This decomposition reveals whether the problem is *queue access* (not getting filled) or *selection* (getting filled at the wrong times).

3.1.11 Microstructure “gotchas” that show up only in production

A few execution realities tend to surprise first-time on-chain CLOB builders:

- **A fill can arrive while a cancel is “in flight.”** Your local state machine must tolerate receiving fills after you have decided to cancel but before you observe confirmation. You must reconcile against the authoritative fill stream rather than assuming intent equals outcome.
- **Partial fills change your risk immediately.** After any partial fill, recompute available margin headroom, inventory limits, and hedge needs. Do not wait for a full fill event.
- **Queue estimates are uncertain; trade accordingly.** Treat Q_{ahead} as an interval $[Q^-, Q^+]$. If you require fills by a deadline, plan an escalation path from passive to IOC/taker-style execution before the deadline arrives.

3.1.12 A compact playbook

Putting the mechanics together, a disciplined passive execution loop on Hyperliquid typically follows these principles:

- **Measure queue, do not guess.** Maintain a live estimate of Q_{ahead} and a fill-rate proxy λ .
- **Protect time priority.** Avoid cancel/re-enter at the same price unless a risk threshold is breached.

- **Adapt to toxicity.** When post-fill markouts become systematically negative, reduce passive size, widen quotes, or switch to controlled aggressive execution.
- **Audit maker/taker behavior.** Log crossed per fill and analyze maker and taker performance separately.
- **Engineer for races.** Your code must reconcile state from fills, not from what you attempted to do.

These ideas set up the next step: the exact semantics of Hyperliquid order types and parameters. Once you can predict how different instructions behave in the book, you can deliberately choose between waiting in the queue, stepping ahead, or taking liquidity when the market state demands urgency.

3.2 Core Order Types

The previous section treated execution as a queueing and selection problem: your PnL depends not only on *where* you trade (price), but *how* you arrive there (resting versus crossing) and *when* your trade occurs (the flow regime that fills you). This section makes that operational by defining the behavioral semantics of Hyperliquid's foundational order types and the most important parameters that control their execution.

Hyperliquid supports the following order types: **Market**, **Limit**, **Stop Market**, **Stop Limit**, **Take Market**, **Take Limit**, **Scale**, and **TWAP**. We will focus on Market/Limit/Trigger orders here; Scale and TWAP are deferred to the advanced execution section.

3.2.1 A precise vocabulary: instruction, evaluation, and outcome

An order is an *instruction* you send. The exchange then evaluates that instruction against the current book and risk checks, producing an *outcome* that you experience as one of:

- **Immediate execution** (fully or partially) against resting liquidity.
- **Resting liquidity** (the order becomes part of the book queue).
- **Cancellation** (because you requested it, because time-in-force demands it, or because constraints prevent resting).
- **Rejection** (invalid parameters or constraints that cannot be satisfied).

Keeping these separate helps you debug bots: many production bugs come from assuming that “submitted” implies “resting,” or that “triggered” implies “filled.”

3.2.2 Market orders: deterministic immediacy, non-deterministic price

Semantics.

A Market order is an instruction to execute immediately against the best available prices on the opposite side of the book until your requested size is filled (or until venue/risk constraints stop it). You control *urgency* (immediacy) and *size*, but you do not directly control the final price path.

What you are buying when you use a Market order.

You are buying two things:

- **Latency robustness.** If your signal decays quickly (breakouts, liquidations, fast mean reversion), waiting for passive fills may be more expensive than crossing.
- **Completion probability.** For many strategies, the cost of not being filled is larger than the cost of paying the spread.

Primary risks.

- **Slippage.** The execution price can walk the book, especially in thin markets or during volatility spikes.

- **Impact and information leakage.** Your order consumes visible liquidity; other participants infer urgency from it.
- **Partial execution under constraints.** If constraints stop execution (risk checks, insufficient liquidity within allowable bounds), you may end with an incomplete fill and residual risk.

Strategic deployment.

Market orders are best reserved for situations where your expected edge over a very short horizon is larger than the spread plus expected slippage. A practical way to encode that into a bot is to require a minimum “signal-to-cost” ratio:

$$\mathbb{E}[\alpha_\tau] > \frac{s}{2} + \mathbb{E}[\text{slippage}].$$

When this inequality is not comfortably true, you generally want to shift toward limit-style execution.

3.2.3 Limit orders: price control, queue uncertainty

Semantics.

A Limit order specifies a maximum buy price (or minimum sell price). It can either:

- **Rest** on the book (providing liquidity) if it is not immediately marketable, or
- **Execute immediately** (fully or partially) if it is priced through the opposite best price (i.e., it crosses the spread).

So a limit order is not inherently “maker” or “passive”; it becomes passive only when its price does not cross.

Time-in-force (TIF) is the real behavioral switch.

On Hyperliquid, time-in-force options include:

- **GTC** (good-til-cancel): the order can rest until filled or explicitly cancelled.

- **IOC** (immediate-or-cancel): any portion not immediately executed is cancelled.
- **ALO** (add-liquidity-only; post-only): the order must not take liquidity; it will rest or be cancelled rather than crossing.

Thinking in terms of execution intent:

- **Limit + GTC** is your default passive tool: you accept queue risk to potentially earn spread/rebates.
- **Limit + IOC** is “controlled aggression”: you attempt to cross up to a price cap, but you refuse to rest.
- **Limit + ALO** is “maker-only”: you refuse to cross even if the book moved and your limit would otherwise execute immediately.

Even if you plan to introduce more complex tactics later (inventory-aware quoting, dynamic spreads), these three combinations are the core building blocks.

Strategic deployment.

Limit orders are appropriate when:

1. Your edge is **not purely speed-based**, so waiting is acceptable.
2. You have a **strong view on price bounds** (e.g., “I will only buy at or below p ”).
3. You explicitly want to **avoid impact** by providing liquidity instead of consuming it.

The trade-off is queue position and selection risk, which you quantified in the prior section. In practice, you will often place limit orders with a timeout and an escalation path: rest briefly, then switch to IOC or market if you have not achieved enough fill by a deadline.

3.2.4 Trigger orders: Stop vs Take, Market vs Limit

Trigger orders add one more layer: they are *conditional instructions*. They sit idle until a trigger condition is met, at which point they submit an *actual* order (market or limit) to the book. Hyperliquid supports **Stop Market, Stop Limit, Take Market, and Take Limit**.

3.2.5 Stop versus Take: what is the condition?

Although both are triggers, their intent differs:

- **Stop** orders are typically risk controls: they activate when price moves *against* your position beyond a threshold.
- **Take** orders are typically profit-taking controls: they activate when price moves *in your favor* to a target.

For a *long* position:

- Stop triggers below entry (or below a risk threshold).
- Take triggers above entry (or above a profit threshold).

For a *short* position, the direction reverses.

This distinction is not cosmetic: in fast markets, stop triggers often occur during *expanding volatility*, while take triggers often occur during *momentum continuation* or *reversions from extremes*. The flow regime at trigger time affects slippage and fill completion.

3.2.6 Market versus Limit after trigger: certainty versus control

Stop Market / Take Market.

- **Goal:** maximize execution certainty once the trigger is hit.

- **Risk:** slippage can be large exactly when you are most stressed (stop-outs in fast drops, take-profits in fast rallies).

Stop Limit / Take Limit.

- **Goal:** cap the worst execution price by posting a limit at (or beyond) a specified level.
- **Risk:** non-execution. In a gap-like move, your limit may never trade, leaving you with the very exposure you were trying to remove.

A useful mental model is:

Trigger Market $\Rightarrow$ execution certainty, price uncertainty

Trigger Limit $\Rightarrow$ price control, execution uncertainty

For systematic trading, the right choice depends on whether the trigger is primarily a *risk bound* (favor market) or a *price target* (limit may be acceptable), and how costly non-execution is in your strategy.

3.2.7 Triggers as part of a position lifecycle

Triggers are often used in pairs to define a “bracket” around a position: one stop-loss and one take-profit. In bot design, treat these as part of the *position state*, not as independent orders, because:

- When the position size changes (partial entry fills, scale-ins, scale-outs), the trigger sizes should be updated to remain consistent.
- If a trigger executes, the opposing trigger should typically be cancelled (unless you intentionally want to reverse or re-enter).

The mechanics of safely constraining these behaviors (e.g., preventing accidental position flips) are handled cleanly with protective flags in the next section, but the strategic point starts here: triggers are not just emergency exits; they are a programmable way to encode your strategy’s decision boundaries into the venue.

3.2.8 Order taxonomy: intent-to-mechanics map

The table below summarizes how these order types map to execution intent and typical use cases.

Instruction	Primary intent	Typical deployment
Market	Immediate execution	Enter/exit when signal is short-lived; emergency risk reduction; completion matters more than price precision.
Limit (GTC)	Provide liquidity at a chosen price	Passive entry/exit; market making; reducing impact; accepting queue and selection risk.
Limit (IOC)	Cross up to a price cap, otherwise do not rest	“Pay up to p ” execution; short deadline execution; controlled aggression without leaving stale resting orders.
Limit (ALO / post-only)	Ensure maker-only behavior	Quotes that must not take; avoid crossing due to stale pricing; capturing maker economics while controlling execution role.
Stop Market	Triggered risk exit/entry with maximum certainty	Stop-losses; breakout entries where missing the move is worse than slippage.
Stop Limit	Triggered order with capped worst price	Stops in calmer regimes; entries/exits where price bounds matter; risk of not filling in fast moves.
Take Market	Triggered profit-taking with high completion	Fast momentum take-profits; de-risking during spikes.
Take Limit	Triggered profit-taking at target price	Profit-taking with price precision; may miss fills in run-away moves.

3.2.9 API-level semantics: the minimal fields you must reason about

At the API level (regardless of client library), the important point is that an order payload typically contains:

- **Direction:** side (buy/sell).
- **Size:** quantity.
- **Price:** for limit-style orders.

- **Risk constraint flags:** e.g., reduce-only (covered next section).
- **Order-type object:** specifying whether it is a limit or trigger order and the relevant parameters (including TIF for limit orders).

A simplified sketch (field names vary by gateway and client) looks like:

```
{
  "asset": 0,
  "side": "buy",
  "size": "1.25",
  "price": "100.0",
  "reduce_only": false,
  "order_type": {
    "type": "limit",
    "tif": "GTC"
  }
}
```

For trigger orders, the key difference is that the `order_type` portion encodes a trigger condition and the kind of order to submit when triggered (market or limit). Even if you do not implement triggers on day one, design your internal order model so that “order intent” (enter/exit, urgency, risk bound) is not hard-coded to a single order primitive; it should compile into one of these types depending on market state.

3.2.10 Choosing the right primitive: three decision questions

When you select among market, limit, and trigger orders, you are answering three questions:

1. **Do I need this executed now, or can I wait?** If now, choose market or IOC-style behavior. If you can wait, choose GTC or post-only quoting.
2. **Is the primary constraint price or completion?** If completion, lean market/trigger-market. If price, lean limit/trigger-limit.
3. **Is this order discretionary or conditional?** If conditional

on a future level, use stop/take triggers; otherwise submit an immediate order.

This framing keeps the logic clean in automated systems: strategy produces *intent*, and an execution policy selects the appropriate order primitive and parameters. The next section tightens that loop by introducing protective flags (notably post-only/ALO and reduce-only) that prevent common failure modes when markets move quickly or when your bot's state lags the venue.

3.3 Protective Order Flags

The core order types in the previous section (market, limit, stop/take) determine *what* you want to do: trade now, trade at a price, or trade only after a condition. Protective order flags determine *what you refuse to do*. For automated execution on a fast on-chain book, these “negative constraints” are often the difference between a strategy that is merely unprofitable and one that is operationally unsafe.

Hyperliquid exposes two especially important constraints:

- **Post-only** via **ALO** (add liquidity only): the order must rest as maker-only and must not execute immediately on placement.
- **Reduce-only**: the order may only reduce an existing position; it must not increase exposure or flip you into the opposite direction.

Both constraints are deterministic and testable. Your job as a trader/builder is to use them to make the exchange enforce your intent even when your bot is wrong, stale, or racing the market.

3.3.1 Post-only (ALO): maker-only as an enforceable constraint

Semantics. A post-only order (Hyperliquid: **ALO**) is a limit order with an additional rule: it must *not* remove liquidity. If its price would cross the current opposite best price (and thus trade immediately), the

venue will not allow it to take; instead, it will be cancelled/rejected according to venue behavior. The practical result is simple:

ALO guarantees you will not accidentally become a taker on entry.

Why ALO matters in real systems. Many execution loops compute a quote price from a stale book snapshot, then submit a limit order. By the time the order arrives, the book may have moved. Without ALO, a “passive” quote can become marketable and cross the spread, turning you into a taker precisely when you least expect it (and often at the worst moment, during volatility).

ALO converts this failure mode from an *unexpected fill* into an *no-fill outcome*. This is almost always preferable for market making and queue-position tactics because your PnL model (rebates, spread capture, selection risk) assumes maker fills.

ALO is not free. The cost is opportunity: if the market moves into your price, ALO can prevent you from getting an immediate fill that might have been acceptable. The right view is that ALO is not about maximizing fills; it is about *preserving role* (maker versus taker) so that your economics and risk estimates remain valid.

Operational guidance. Use ALO for:

- market-making quotes that should never cross,
- inventory-rebalancing quotes where taking is not desired,
- any time you require maker behavior for fee/rebate or for microstructure reasons.

Avoid ALO when:

- you genuinely want a fill now and are willing to cross,
- your logic already uses IOC/market for urgency, making the maker-only guarantee unnecessary.

3.3.2 Reduce-only: exposure cannot increase, even if you mis-size

Semantics. Reduce-only is a first-class order option: it reduces an existing position instead of opening a new position in the opposite direction. This is a constraint on *net position change*. Conceptually, if your current position is P (positive for long, negative for short) and the order would change position by ΔP , then reduce-only enforces:

$$|P + \Delta P| \leq |P|$$

and

$$\text{sign}(P + \Delta P) \in \{\text{sign}(P), 0\}.$$

In words: the order may move you toward flat, and may flatten you, but must not push you past zero into a flipped position.

The API schema exposes reduce-only as a boolean flag (e.g., `r` in some gateway schemas). Treat this as a *risk control*, not as a convenience.

Why reduce-only matters: the position-flip accident. Position flips are more common than they sound, particularly in these cases:

- **Oversized exits.** Your bot thinks you are long 1 contract, but due to partial fills or a reconciliation lag you are long 0.4. If you send a sell for 1 without reduce-only, the extra 0.6 becomes a new short.
- **Bracket orders with moving size.** You enter in pieces, but your take-profit/stop-loss sizes were created from the intended size rather than the realized size.
- **Retry logic.** A network hiccup triggers a retry of an exit order that already partially filled.

Reduce-only turns these from “flip and hope you notice” into “flatten at most.” This makes downstream logic (hedging, leverage control, liquidation avoidance) far safer.

Reduce-only and partial fills. Reduce-only does not mean “all-or-nothing.” If you are long $P > 0$ and submit a reduce-only sell of size

q , the exchange can fill any amount up to $\min(q, P)$. If liquidity is thin, you may be partially reduced and remain exposed.

This is exactly what you want: the venue prevents the unintended sign change, but it cannot promise immediacy. Your state machine must therefore handle sequences like:

```
position: +1.00  
send reduce-only sell 1.00  
fills: 0.30, 0.20  
position: +0.50  
(cancel/replace or wait)
```

The key is that every fill moves you closer to flat, never beyond.

3.3.3 Combining ALO and reduce-only: constrained execution intent

These flags can be combined with different intents:

- **ALO without reduce-only:** typical for quoting and market making. You are willing to add exposure, but only via maker fills.
- **Reduce-only without ALO:** typical for risk exits. You insist on not increasing exposure, but you may accept taker behavior to get out.
- **ALO + reduce-only together:** less common but useful when you want to *de-risk passively* (e.g., scale down inventory with maker fills), accepting that you may not get filled quickly.

Thinking this way helps you avoid mixing incompatible goals. For example, “I need out now” and “I must be maker-only” cannot both be guaranteed; ALO will preserve role, not urgency.

3.3.4 A logic tree for how the venue enforces these constraints

The following logic is the practical execution contract you should assume: the venue evaluates the order’s marketability (would it cross?)

and the order's effect on your position (would it increase or flip?), and then either allows execution/resting or returns a cancel/reject outcome depending on constraints.

3.3.5 Common edge cases and how to defend against them

3.3.6 Stale position state when sending exits

The most frequent reduce-only failure is not conceptual; it is operational: your bot's view of position P is stale due to partial fills, missed messages, or out-of-order event processing. Two best practices:

- **Always mark exits reduce-only** unless you explicitly want to reverse.
- **Update position from fills, not from order acknowledgements.** Acknowledgements confirm submission, not execution.

Reduce-only is the venue-enforced backstop that makes this robust.

3.3.7 Post-only rejections due to microprice drift

ALO orders can fail simply because your computed quote crosses by a tiny amount after the book moves. This is not an error condition; it is a normal outcome of insisting on maker-only. Treat it as a signal:

- If ALO fails frequently, your quote logic is too tight for current volatility or your update cadence is too slow.
- The correct response is usually to reprice (widen) or pause quoting briefly, not to drop ALO.

3.3.8 Trigger orders that behave like markets during stress

A stop or take trigger that submits a marketable order can behave like a market order at the worst time, which is why you need a separate

risk decision: do you want *certainty of exit* (trigger market) or *price bounds* (trigger limit)? Regardless of that choice, reduce-only remains critical for any trigger meant to close or reduce an existing position.

3.3.9 Monitoring: confirming your protections worked

Protective flags should be validated continuously, not assumed. Hyperliquid's fill streams provide two particularly useful checks:

- **Maker vs taker classification.** If your passive strategy uses ALO, you should observe maker-side fills (e.g., `crossed=false`) for those orders. Unexpected `crossed=true` events are a red flag that your order flow is not constrained the way you think (or that some orders are not ALO).
- **Liquidation detection.** Fill streams allow detection of forced execution conditions; liquidation-related fills can be identified and are taker-side for the liquidated user. If you see these, treat them as a post-mortem trigger: your protective exits were late, too small, or not reduce-only in a way that aligned with your intended risk bounds.

In production, these checks should become automated alerts. The purpose of protective flags is not just to reduce losses; it is to make system behavior auditable under stress.

3.3.10 Summary: constraints as part of strategy design

Post-only (ALO) and reduce-only are not minor toggles; they are contract clauses you ask the venue to enforce:

- ALO ensures you do not pay the spread by accident when you intended to quote.
- Reduce-only ensures exits cannot silently become reversals when your sizing or state is wrong.

With these constraints in place, you can safely build more complex execution logic—including native algorithmic orders and schedule-based execution—without turning every network race or partial fill into a risk event.

3.4 Advanced Execution Orders

Advanced execution orders sit one level above market, limit, trigger, ALO, and reduce-only orders. They are *meta-orders* that break a large desired trade, or parent order, into many smaller child orders over time and across price levels. The goal is to reduce impact, reduce selection risk at any single moment, and make execution quality more predictable.

Hyperliquid supports **Scale** and **TWAP** orders natively. Even if you ultimately write your own execution algorithm, understanding the venue-native semantics matters because native orders have deterministic behavior, and on-chain transparency creates special considerations about information leakage.

3.4.1 Why slicing works: impact is convex in size

Large marketable orders tend to have *nonlinear* cost: doubling size often more than doubles the average slippage because you traverse deeper levels of the book and induce others to reprice. A simple practitioner model decomposes execution cost into:

$$\text{Cost} \approx \text{spread} + \text{temporary impact} + \text{opportunity risk}$$

Slicing reduces temporary impact by trading smaller clips that are closer to the book's natural replenishment rate. The trade-off is opportunity risk: slower execution leaves more time for the market to move away. This is the same impact-risk tension formalized in the Almgren-Chriss framework: higher urgency increases expected cost through impact, while lower urgency increases variance through market risk during execution.

TWAP is the simplest urgency schedule: it chooses a constant trading rate over a fixed horizon. Scale orders are the simplest price schedule: they place multiple limits across a range of prices to avoid crossing too aggressively.

3.4.2 Venue-native TWAP: deterministic cadence and constraints

What TWAP means on Hyperliquid.

Hyperliquid TWAP is venue-native with explicit mechanics: child orders are sent every **30 seconds**. Each suborder has a **max slippage of 3%**. If execution falls behind schedule, catch-up behavior is capped at **3x** the normal slice.

You should read these as hard constraints on the realized child-order flow:

- **Cadence of 30 seconds** defines the shortest timescale at which the TWAP can adapt. If conditions change dramatically between ticks, the TWAP will still wait for the next tick to submit another child.
- **Max slippage of 3% per child** is a safety valve against extreme price jumps, but it also introduces *shortfall risk*: if the market runs away beyond the slippage band, that child may not execute.
- **Catch-up cap of 3x** prevents the TWAP from suddenly becoming a very large aggressive order when it is behind, but it also means it may remain behind for a while in fast markets.

What you control.

At a high level you control the parent's total size and the total time horizon. From those, the venue determines an implied baseline slice size per 30-second interval. Conceptually, if you want to buy total quantity Q over horizon T seconds, the baseline number of slices is:

$$N = \left\lceil \frac{T}{30} \right\rceil,$$
$$q_{base} \approx \frac{Q}{N}.$$

The realized slice q_k at tick k is then shaped by how much has already executed, the 3% slippage constraint, and the 3x catch-up limit.

3.4.3 TWAP failure modes you must plan for

TWAP is not set and forget; it is set and monitor. The three main operational outcomes to plan for are:

1) Underfill due to slippage constraints.

If the market moves away quickly, a child may refuse to trade beyond its 3% limit. Your parent schedule then accumulates unfilled quantity. This is not a bug; it is a consequence of capping worst-case execution.

2) Toxic catch-up in trending markets.

If the TWAP falls behind during a trend, catch-up will increase clip sizes, up to 3x baseline. Catch-up clips arrive exactly when the market has already demonstrated momentum, which can make those clips expensive. The cap prevents pathological behavior, but it does not eliminate the issue.

3) Predictable footprints.

A 30-second deterministic rhythm can be detected statistically. Even if others cannot see your intent directly, periodic child prints can be used to infer ongoing execution, and some counterparties will adapt by widening or stepping away.

These are not reasons to avoid TWAP; they are reasons to decide *what TWAP is for*. TWAP is best when:

- you can tolerate completing over a fixed horizon rather than immediately,
- you want to bound per-child slippage,

- you are willing to accept some completion risk in exchange for lower impact.

3.4.4 Transparency and sunshine trading risk

On-chain venues make parts of order flow more observable. Recent research frames Hyperliquid TWAP as a form of sunshine trading because terms can be visible from inception, creating the possibility of information leakage: others can anticipate future child demand and reprice in advance.

The important takeaway for practitioners is tactical:

- If your execution intent is predictable and visible, the market can move from being an *execution venue* to being an *adversary*. Liquidity providers may widen; arbitrageurs may position ahead; short-term traders may trade in front of your schedule.
- The cost is paid as worse average prices, worse fills on passive children, and higher slippage on aggressive children.

This does not imply that native TWAP is always inferior. It implies that you should not use it for *alpha-sensitive* execution where concealing intent is part of your edge. In those cases, a custom schedule that is less detectable, with variable timing, randomized clips, and conditional pausing, may dominate, even if it is less convenient.

3.4.5 Connecting TWAP knobs to the Almgren-Chriss view

The Almgren-Chriss framework is a canonical way to think about optimal liquidation: you choose a trading trajectory that minimizes a combination of expected cost, through impact, and risk, through variance of price moves during execution. You do not need the full derivation to use the insight:

- A **longer horizon** reduces instantaneous impact but increases exposure to price risk while you wait.

- A **shorter horizon** reduces market risk but increases impact.

TWAP corresponds to a constant rate. In Almgren-Chriss terms, it is a simple heuristic for a moderate risk-aversion setting where you do not want to concentrate trades too heavily at the start or end.

A practical method to choose a TWAP horizon T is to start from a risk budget:

$$R_{\text{execution}} \approx \sigma \sqrt{T} \cdot P_{\text{remaining}}$$

Here, $P_{\text{remaining}}$ represents the position not yet executed. Adjust T until that risk is acceptable relative to your allowed drawdown or hedge capacity. Then check whether the implied clip sizes are reasonable relative to observed market depth and recent trade volume.

3.4.6 Order-flow-aware schedules: when constant rate is not optimal

Cartea-Jaimungal-style execution highlights a simple idea: if you can predict near-term net order flow, you can trade *more* when the market is likely to go your way and *less* when you would be run over.

In practical perps execution language:

- If you are buying and you expect net buying pressure, or upward momentum, you may want to **accelerate** to avoid chasing.
- If you are buying and you expect net selling pressure, or downward mean reversion, you may **slow down** and let liquidity come to you.

Venue-native TWAP cannot incorporate these signals beyond its fixed cadence and constraints. A custom execution algorithm can: it can pause, speed up, flip between passive and aggressive children, or dynamically change price caps. This is one of the main reasons sophisticated desks use TWAP as a baseline but not as an endpoint.

3.4.7 Scale orders: distributing a limit schedule across prices

Scale orders provide another kind of impact control: rather than trading at a constant rate over time, you distribute liquidity across a range of prices. Conceptually, a scale order is equivalent to submitting multiple limit orders $\{(p_i, q_i)\}_{i=1}^K$ where prices p_i span a band and sizes q_i sum to a total desired quantity.

Why scaling can reduce impact.

If you want to buy Q , a single aggressive trade pays the spread and walks depth. A scale schedule can:

- **Let the market come to you** on part of the size, using resting bids,
- **Avoid signaling urgency** because you are not repeatedly crossing,
- **Control worst price** by limiting the highest price you will pay.

The trade-offs.

Scale orders exchange impact for fill uncertainty:

- You may not complete if the market never trades down into your bid ladder for buys, or up into your ask ladder for sells.
- If price moves quickly through your ladder, you can be filled aggressively at multiple levels, potentially increasing selection risk.

A simple scale design pattern.

A common pattern is to choose prices at fixed tick or percentage intervals and to weight sizes to reflect your urgency. For a buy schedule across K levels:

$$p_i = p_0 - (i - 1)\Delta p, \quad i = 1, \dots, K$$

Choose q_i either uniform, with $q_i = Q/K$, for neutral urgency, or front-loaded or back-loaded depending on whether you want more size closer to the market.

When you pair scale orders with **ALO**, you can ensure the ladder does not accidentally cross and become taker flow. When you pair exits with **reduce-only**, you can ensure the ladder cannot flip your position if your realized position is smaller than intended.

3.4.8 Native versus custom: a decision checklist

You can treat venue-native TWAP and Scale orders as reliable execution appliances, and custom algorithms as flexible execution software. The right choice depends on constraints:

Constraint or goal	Venue-native TWAP or Scale	Custom execution algorithm
Deterministic behavior and simplicity	Strong: fixed cadence and defined slippage and catch-up rules	Depends on your implementation quality
Information leakage sensitivity	Potentially weak if terms are visible and predictable	Can randomize timing and size; can hide intent better
Adaptation to order flow and volatility regimes	Limited because rules are fixed	Strong: can pause, accelerate, and change aggressiveness
Operational burden: state, retries, and monitoring	Lower because the venue handles slicing	Higher; you own edge cases and reconciliation

3.4.9 Monitoring TWAP and Scale progress

Even with native algorithms, your system must track progress and detect when execution quality deviates from plan. Vendor data stream catalogs include **TWAP-specific streams** that can support analytics and monitoring of TWAP progress and slippage. At minimum, you should compute:

- **Executed quantity** versus schedule: are you ahead or behind?
- **Average execution price** versus arrival price, also known as implementation shortfall.
- **Per-child slippage and reject or no-fill rates**: are constraints binding?
- **Role mix**, meaning maker versus taker, using fill fields such as crossed.

If you detect systematic underfills, you need a policy decision: extend the horizon, allow more aggressive execution, such as switching to IOC or market for the remainder, or accept partial completion.

3.4.10 Advanced orders as controlled exposure to the book

TWAP and Scale orders are systematic ways to turn a large discretionary action into many small, rule-governed actions:

- **TWAP** controls *time*: it spreads execution across a fixed horizon with a 30-second cadence, bounded by a 3% max slippage per child and a 3x catch-up limit.
- **Scale** controls *price*: it distributes size across a ladder to reduce immediate impact and to bound worst price.

The key strategic trade-off is always the same: lower impact and smoother execution versus completion risk and, on transparent venues, potential information leakage.

3.5 Execution Tactics

The earlier sections established the mechanical levers you can pull (market/limit/trigger; GTC/IOC/ALO; reduce-only) and the economic forces that punish naive use (queue uncertainty, selection risk, impact). This section turns those pieces into a practical execution loop: a small set of real-time heuristics coupled to a robust state machine that can survive partial fills, racing messages, and changing market conditions.

The objective is not to “win every tick.” The objective is to (i) get the intended exposure with bounded cost, (ii) avoid unintended role changes (maker vs taker), (iii) prevent position flips, and (iv) keep internal state synchronized with what the venue actually did.

3.5.1 The two-mode view: passive capture vs aggressive completion

Most production execution systems can be understood as a controller that switches between two modes:

- **Maker mode (passive):** place resting liquidity (often with ALO) to capture spread/rebates when conditions are benign.
- **Taker mode (aggressive):** cross the spread using market or IOC-style behavior when urgency dominates.

The switch is triggered by *state* (how much is already filled, time remaining, inventory constraints) and *microstructure signals* (spread, volatility, one-sided flow).

A clean way to encode this is to define an *urgency score* $U \in [0, 1]$ that increases when you need completion and decreases when passive capture is safe. Then choose: if $U < U_{maker}$ then Maker mode, if $U > U_{taker}$ then Taker mode, with hysteresis $U_{maker} < U_{taker}$ to prevent oscillation.

3.5.2 Microstructure heuristics that actually map to decisions

Heuristics are useful only when they directly map to actions. The list below is intentionally short; each item corresponds to a switch in your execution policy.

3.5.3 Spread and “paying up” threshold

If you cross, you pay at least half the spread plus slippage. Therefore, your system should know the current spread s and enforce a maximum acceptable crossing cost. A simple rule:

- If s is **small and stable**, aggressive execution is cheaper (taker mode more acceptable).

- If s is **wide or unstable**, passive execution tends to dominate unless you face a hard deadline.

Implementation detail: compute s from top-of-book updates; add a “spread shock” flag if s exceeds a rolling percentile of its last few minutes.

3.5.4 Volatility/markout filter for maker exposure

From Section *Matching Priority and Fill Dynamics*, you know the key maker risk is negative post-fill drift. A practical online proxy is a rolling “maker markout” computed on your own maker fills (identified with `crossed=false` in the fill stream):

$$\hat{D}(\tau) = \text{EWMA}(\text{side}(m_{t+\tau} - m_t))$$

Execution policy mapping:

- If $\hat{D}(\tau)$ becomes strongly negative at short horizons, throttle maker size, widen quotes, or switch to taker mode for completion.
- If $\hat{D}(\tau)$ is near zero/positive, maker mode is behaving as intended.

This is one of the few signals that is directly aligned with your realized PnL mechanics.

3.5.5 Order-flow imbalance (toxicity proxy)

In competitive perps, the fastest way to lose as a maker is to quote into one-sided aggressive flow. A simple imbalance proxy is:

$$I = \frac{V_{buy} - V_{sell}}{V_{buy} + V_{sell}}$$

over a short window (e.g., 5–30 seconds), where V_{buy} and V_{sell} are aggressive volumes inferred from trade prints.

Policy mapping:

- If you are providing asks (selling) and $I \gg 0$ (aggressive buying), reduce ask exposure or move further from mid.
- If you are providing bids (buying) and $I \ll 0$ (aggressive selling), reduce bid exposure or move further from mid.

This is an execution-level version of order-flow-aware control: you slow down in the direction that is currently toxic and speed up when flow is favorable.

3.5.6 Deadline-driven escalation: passive first, then controlled aggression

A widely effective tactic is a **hybrid schedule**:

- Start passive (GTC/ALO limit) to attempt cheap fills.
- If not filled by a deadline, switch to IOC (or market) with a price cap.

This is an engineering-friendly translation of optimal execution theory: you attempt to earn liquidity provision benefits early, then buy completion later as the opportunity cost of waiting increases.

A minimal policy uses remaining time T_{rem} and remaining quantity Q_{rem} :

$$\text{target participation rate} \approx \frac{Q_{rem}}{T_{rem}}.$$

If your realized fill rate in maker mode falls below this target for too long, you escalate.

3.5.7 Order management: robust loops without self-inflicted queue loss

Even excellent signals fail if your order management is brittle. Two patterns dominate production reliability.

3.5.8 Avoid unnecessary churn: when to update a quote

Every quote update costs you time priority and introduces race conditions. Use thresholded updates:

- **Reprice only if mid moves** beyond a tolerance band (e.g., more than k ticks).
- **Reprice only if spread regime changes** (wide $\rightarrow$ narrow, narrow $\rightarrow$ wide).
- **Reprice if inventory constraint changes** (you must de-risk).

If none of these triggers fire, prefer to rest and let queue position mature.

3.5.9 Controlled aggression: IOC rather than full market when possible

When escalating, IOC-style limit behavior often dominates pure market orders:

- You accept immediate execution if available up to a cap.
- You avoid leaving stale resting orders that might fill later when your state has changed.

This blends urgency with price control. It is also simpler to reason about in code: the order either fills now (fully/partially) or disappears.

3.5.10 State management: reconcile from fills, not from intent

Your internal state must reflect what the venue did, not what you attempted.

3.5.11 Use the fill stream as the execution source of truth

Hyperliquid provides fill streams where maker vs taker can be identified via `crossed=false` (maker) and `crossed=true` (taker). Treat each fill as the atomic event that updates:

- position and average entry,
- realized fees/rebates (fee sign can corroborate maker/taker role),
- remaining quantity for the parent objective,
- and risk limits (margin headroom, max inventory).

Never update these from order submission acknowledgements alone.

3.5.12 Handle partial fills as a first-class state

Every order should be modeled with a remaining quantity q_{open} , not a boolean status. Your parent objective likewise should track Q_{rem} . This avoids the most common bug class: assuming “filled” when you are merely “partially filled,” and sending an exit that is too large (which reduce-only then has to save you from).

3.5.13 Sync and resync: surviving missed messages

On-chain and streaming systems can drop messages or reorder them. Your loop needs a periodic reconciliation step:

- refresh open orders,
- refresh current position,
- compare to internal state,
- resolve discrepancies by trusting venue state and replaying missing deltas.

If you cannot name a specific API call for this in your stack, implement it in the most conservative way available: pause new submissions, pull authoritative state, rebuild internal books/positions, then resume.

3.5.14 Putting it together: a minimal execution policy template

The following template is intentionally generic and composes the elements covered so far:

- **Define parent intent:** target side, total size Q , deadline T , and whether reversal is allowed.
- **Choose safety constraints:** Use **reduce-only** for exits and any order that must not flip you. Use **ALO** for passive quotes that must not take.
- **Initialize in maker mode** with a passive limit (often ALO) sized to a safe inventory increment.
- **Monitor:** fill events, spread regime, volatility proxy, and imbalance proxy.
- **Escalate if needed:** if behind schedule or toxicity spikes, switch to IOC/taker mode for a portion of Q_{rem} .
- **Reconcile regularly:** resync open orders and positions; repair internal state.
- **Terminate cleanly:** once $Q_{rem} \approx 0$ or the deadline passes, cancel remaining orders and ensure state is flat (if required).

This template becomes powerful when paired with a precise state machine that prevents contradictory actions (e.g., sending a new order while a cancel is unresolved) and that cleanly handles partial fills.

3.5.15 Capstone: a production-style execution state machine

A good execution state machine is not about complexity; it is about *unambiguity*. At any moment, you should be able to answer:

- What orders does the venue currently consider open?
- What is my remaining parent quantity?
- Am I allowed to submit a new order, or must I wait for a cancel/ack?
- Did I just get filled as maker or taker, and does that change my mode?

The following state machine is a practical blueprint. You can implement it in any language; the key is to preserve the transitions and the invariants.

3.5.16 Implementation notes: invariants to enforce in code

To make the above state machine real, enforce these invariants:

- **Single source of truth for fills:** position and Q_{rem} only update from fill events.
- **No overlapping intent:** do not submit a new child while an old child is in an unresolved cancel/ack state, unless your design explicitly supports multiple concurrent orders and you can reconcile them safely.
- **Constraint consistency:** maker-mode children should use ALO; exit children should use reduce-only unless reversal is intended.

- **Periodic reconciliation:** a timed resync is mandatory, not optional, in streaming environments.

Effective execution on Hyperliquid is a blend of microstructure awareness and disciplined state management:

- Use simple, measurable signals (spread regime, markout/toxicity, imbalance, deadline pressure) to shift between passive and aggressive behavior.
- Use ALO and reduce-only to make the venue enforce your execution intent under stress.
- Build your bot around a fill-driven state machine that treats partial fills and synchronization as normal operating conditions.

With this structure, you can layer in more sophistication—better predictors, adaptive sizing, and portfolio-level coordination—without breaking the execution core that keeps the strategy safe and testable.

Chapter 4

Market Data, Simulation, and Quant Workflows

In the relentless arena of on-chain perps, intuition is a fragile weapon. True alpha is forged in the crucible of rigorous quantitative research. This chapter bridges the critical gap between raw exchange data and live deployment, equipping you with the infrastructure and simulation frameworks needed to backtest with high fidelity, detect shifting market regimes, and accurately attribute every fraction of a basis point in your PnL. Welcome to the quant's laboratory, where strategies are systematically battle-tested before they ever risk a dollar.

4.1 Market Data Types for Orderbook Strategies

Orderbook strategies live and die on *microstructure correctness*. If your historical reconstruction differs from live trading by even small details—missing a few deltas during a burst, timestamping trades at receive-time instead of exchange-time, or rounding size in a way the

venue would reject—your backtest will “discover” alphas that cannot survive production.

This section defines (i) the core data modalities you must capture on Hyperliquid for high-frequency research, (ii) how to store them so you can replay the market with high fidelity, and (iii) the ingestion architecture that keeps the dataset complete under real-time load. The goal is simple: every backtest should be able to re-run from raw events and reconstruct the same book states your strategy would have observed live.

4.1.1 The core data modalities (minimum viable microstructure dataset)

For an orderbook-driven strategy, you typically need three public data families plus one “metadata” family. A disciplined workflow treats each family as a first-class table/stream with explicit schemas, timestamps, and provenance.

4.1.2 L2 order book: snapshots + incremental deltas

Hyperliquid’s order book stream provides *snapshots plus incremental changes*, which is the correct design for maintaining a local book in real time. Your archive must preserve both:

- **Snapshots:** a full state for a symbol at a point in time. Snapshots are the only robust resynchronization anchor after packet loss, reconnects, or software pauses.
- **Deltas:** incremental updates that transform the prior state into the next state. Deltas carry the microstructure path; for simulation, you need the sequence, not just occasional states.

What you store (practical requirements). For each symbol, store:

- the **raw snapshot payload** (as received) and a normalized representation (e.g., arrays of price levels and sizes),

- the **raw delta payloads** in arrival order,
- a monotonically increasing **local sequence index** you assign upon ingestion (even if the feed has its own ordering, keep your own to enforce deterministic replay),
- **timestamps** at multiple stages (discussed in the Timestamping, ordering, and determinism section).

Depth and precision. Capture the maximum depth your strategy might depend on. Even if your quoting logic only uses top-of-book today, deeper levels are valuable later for:

- estimating *impact* and slippage models,
- identifying hidden liquidity patterns (e.g., replenishment),
- measuring imbalance beyond the best level.

Also archive sizes/prices at full precision consistent with the venue's constraints. Hyperliquid's `/info metaAndAssetCtxs` exposes per-asset trading parameters (e.g., `szDecimals`, `maxLeverage`, `onlyIsolated`) that your research code should treat as *schema constraints*, not afterthoughts.

4.1.3 Trade prints (public): the only ground truth for aggressor flow

Public trades are indispensable even for “pure” passive strategies:

- Trades are the definitive record that price actually exchanged.
- They are the cleanest proxy for *aggressive order flow* (what hit the book), which drives adverse selection and inventory risk.
- They provide cross-checks against your reconstructed book (e.g., “did the best ask size drop because it traded, or because it was canceled?”).

Store trades as an append-only stream with:

- symbol, price, size, side (if provided), and any exchange identifiers in the payload,
- exchange-provided timestamps (if present) and local receive timestamps,
- a per-symbol ingestion sequence index.

4.1.4 Funding series (perps-specific carry and PnL driver)

Perpetuals add a financing leg: *funding*. Funding is not a nuisance variable; it is often a dominant component of PnL and also a regime feature (persistent positive/negative funding can indicate crowded positioning).

For research you need:

- **Funding history** for each asset (via `fundingHistory` on the `/info` surface, per the notes),
- ideally, **real-time funding updates** if you model intraperiod carry or want accurate live-vs-backtest reconciliation.

Funding data should be stored with explicit *effective time* (when the rate applies) distinct from *observation time* (when you learned it). This distinction prevents look-ahead bias.

4.1.5 Metadata and context: `metaAndAssetCtxs` as “market configuration”

Many backtests fail silently because they simulate impossible orders. Hyperliquid’s `metaAndAssetCtxs` combines:

- **Universe metadata:** per-asset parameters such as `szDecimals`, `maxLeverage`, `onlyIsolated`.
- **Asset contexts:** fields such as `markPx`, `midPx`, `oraclePx`, `funding`, `premium`, `openInterest`, `dayNtIVlm`, and `impactPxs`.

Two practical uses:

- **Constraint-aware simulation:** Size rounding, leverage caps, and isolated-only flags must be enforced at *signal time*, not patched later.
- **Coarse execution-cost proxies:** `impactPxs` can approximate slippage when you cannot (or choose not to) replay full depth. Even if you *do* replay depth, `impactPxs` is a useful sanity check.

4.1.6 Public vs. private data: what you need for research vs. what you need for reconciliation

Orderbook alpha research primarily uses public feeds (L2 + trades + funding). However, once you go live, a rigorous workflow also archives private/user event streams to reconcile execution and PnL:

- **order updates** (acknowledgments, state transitions),
- **user fills** (your executions),
- **user funding fee** events (your funding payments),
- **user events** more broadly (account-level events).

These are essential for diagnosing the “reality gap” between your fill model and actual execution, but they are *not* substitutes for public market data; they answer different questions.

4.1.7 Timestamping, ordering, and determinism

High-frequency backtests are usually broken by one of three issues: ambiguous time, inconsistent ordering, or non-deterministic replay.

4.1.8 Use multiple clocks and keep them separate

For each inbound message (book snapshot, book delta, trade, funding update), record at least:

- **exchange timestamp** (if present in payload),
- **receive timestamp** (when it hit your network stack),
- **ingest timestamp** (after parsing/validation, when you committed it).

The exchange timestamp supports research alignment (e.g., joining trades to deltas). Receive/ingest timestamps quantify latency and allow you to replay *as-observed* data (what your strategy could have known at the time).

4.1.9 Assign your own sequence numbers

Even if a venue provides ordering, your pipeline should create a per-stream, per-symbol sequence index on ingestion:

- It makes replay deterministic.
- It allows you to detect gaps.
- It protects you from subtle reordering caused by asynchronous callbacks, batching, or multi-threading.

4.1.10 Dropped-message monitoring is not optional

The Hyperliquid repository notes an operational hazard: `dropped_message_count` can increase when an internal queue fills (callbacks too slow). This is a dataset integrity event. Treat it like a circuit breaker:

- Log the counter (and changes) with timestamps.
- On any increment, mark the affected interval as *tainted* for microstructure backtests unless you can resync from a snapshot boundary and prove continuity.

Without this, your backtest may unknowingly “teleport” the book state across missing deltas.

4.1.11 Storage design: event logs first, derived bars second

A common anti-pattern is to store only candles or periodic snapshots. That is acceptable for slower signals, but insufficient for orderbook strategies. The recommended pattern is:

- **Raw immutable event log (gold):** store every message as received (compressed JSON, protobuf, or equivalent), plus your ingestion metadata (timestamps, sequence, source).
- **Normalized event tables (silver):** parse into typed columns (symbol, price, size, side, etc.) for fast research queries.
- **Derived features (bronze):** build mid-price series, spreads, imbalances, realized vol, etc., *from the normalized events*.

Why immutable logs matter. If a feature looks suspicious months later, you can re-parse the raw stream with a corrected decoder. If you only stored derived tables, you cannot recover what you lost.

4.1.12 Partitioning and indexing for HFT back-tests

Design partitions around how you replay:

- Partition by **symbol** and **date/hour** for L2 and trades.
- Index by **(symbol, ingest_seq)** and **(symbol, exchange_ts)** where applicable.
- Keep snapshots in a separate table keyed by **(symbol, snapshot_seq)** so you can seek to the nearest anchor quickly.

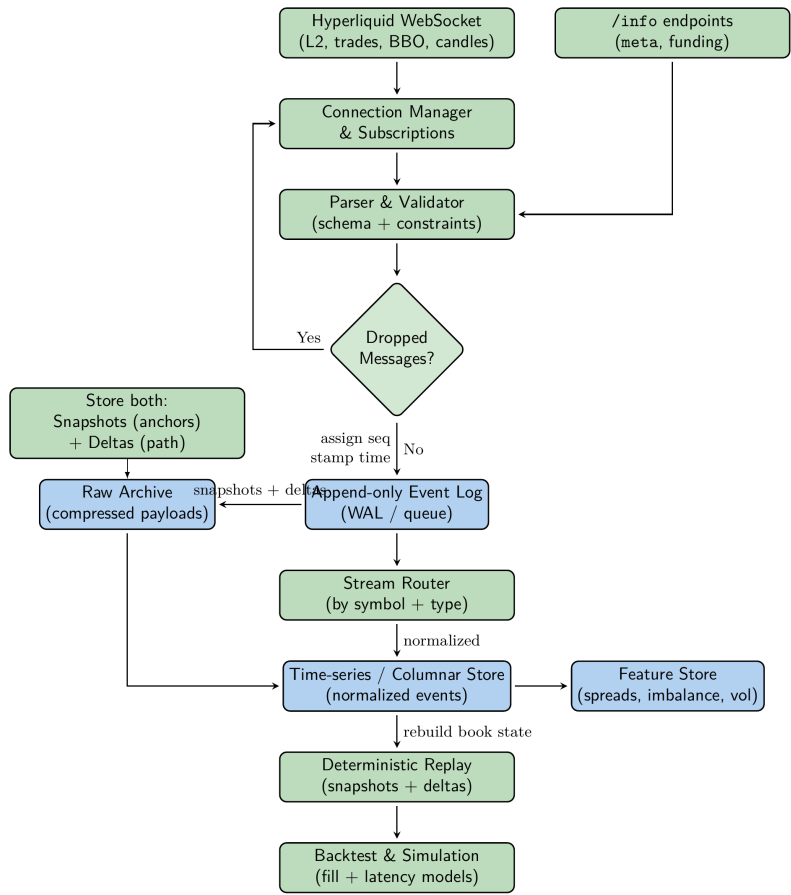
4.1.13 Data retention and compression

Orderbook deltas are high-volume. Practical tactics:

- compress raw payloads (zstd/gzip) and store columnar normalized tables (e.g., Parquet) for research,
- store *every* delta for a shorter window (e.g., 30–180 days) and maintain longer-horizon summaries for regime research,
- always keep snapshots longer than deltas because they permit partial reconstruction and sanity checks.

4.1.14 A robust ingestion pipeline (from WebSocket to research-grade databases)

You want a pipeline that is simultaneously (i) low-latency for live trading and (ii) loss-intolerant for archival. The key is to decouple the *network ingestion path* from the *storage path* using an internal append-only log and explicit backpressure.



4.1.15 Operational checklists for “research-grade” capture

To keep the dataset fit for millisecond-level backtests, implement these checks as automated monitors, not manual habits.

4.1.16 Completeness checks

- **Gap detection:** verify ingest sequence increments without missing values per stream.
- **Snapshot cadence:** enforce a rule such as “every reconnect must begin with a fresh snapshot” and archive it.
- **Drop counter alarms:** alert on any increase in `dropped_message_count` and tag the time range.

4.1.17 Correctness checks

- **Schema validation:** reject or quarantine messages that violate expected types/fields rather than silently coercing.
- **Constraint validation:** cross-check sizes and rounding against `szDecimals` from `metaAndAssetCtxs`.
- **Cross-feed sanity:** confirm that trades are plausible given the reconstructed top-of-book (allowing for latency and update ordering).

4.1.18 Performance checks

- Measure end-to-end ingestion latency (receive → commit).
- Monitor callback/consumer backpressure; the moment your consumer cannot keep up, you are accumulating unbounded risk of drops.

4.1.19 A compact schema blueprint (what to store and why)

Dataset	Granularity	Why it matters for HFT research
L2 snapshots	irregular (on start/reconnect/periodic)	Resync anchors; enable deterministic replay and integrity recovery after gaps.
L2 deltas	event-by-event	Preserve the true microstructure path; required for queue/priority approximations and realistic fill simulation.
Trades	event-by-event	Ground truth of aggressor flow; calibrates adverse selection and validates book reconstruction.
Funding history (fundingHistory)	interval-based	Separates carry from trading alpha; required for funding PnL attribution and regime features.
Universe + context (metaAndAssetCtxs)	periodic polling + on change	Enforces feasibility (size decimals, leverage constraints) and provides context features (OI, premium, impactPxs).
Private user streams (fills/-funding/order updates)	event-by-event	Reconciles live execution and PnL; diagnoses simulator bias and latency effects.

Table 4.1: Research-grade market data blueprint for orderbook strategies on Hyperliquid.

4.1.20 The guiding principle: archive what you would have known at the time

A final discipline that prevents subtle leakage: store data in a way that supports an *as-observed* replay. In production you do not know the future, and you do not receive all streams at the same instant. By retaining receive-time ordering (plus exchange-time fields for alignment) you can run two complementary backtests:

- **As-observed backtest:** replay in the order your system saw events; best for latency-sensitive strategies and engineering validation.
- **Exchange-time backtest:** align by exchange timestamps; best

for estimating structural relationships (e.g., response to flow), while still guarding against look-ahead.

Once you can reconstruct the same book state deterministically—and you can prove your dataset has no silent gaps—alpha research becomes a statistical problem rather than an archaeological one.

4.2 Backtesting and Fill Simulation

A strategy that looks profitable on paper but cannot be executed under realistic conditions is not an alpha; it is a measurement error. For orderbook strategies on perps, the main sources of “paper alpha” are:

- **Unrealistic fills** (assuming you always trade when your quote touches the market).
- **Ignored latency** (assuming orders appear instantly and cancels always win races).
- **Deterministic execution** (assuming the same state always produces the same fill path).

This section lays out simulation architectures that address these three failure modes while staying implementable. The goal is not to perfectly clone the exchange matching engine (you cannot), but to produce a simulator whose biases are *known*, *bounded*, and *stress-tested*.

4.2.1 What “realistic” means for an HFT backtest

A high-frequency backtest is credible when it reproduces, to the degree allowed by your data:

- **Partial fills:** passive orders often fill in pieces, and the time profile of those pieces matters for inventory and hedging.
- **Race conditions:** cancels, modifies, and opposing trades compete; the “winner” is not deterministic.

- **Latency path:** network transit + exchange processing delays imply you trade on a stale view and your actions arrive late.
- **Priority effects:** passive execution depends on how much resting volume is ahead of you and how that volume evolves.

As we saw in the data section, this is why you archive L2 snapshots/deltas and trade prints: without them, you cannot even attempt a microstructure-consistent fill model.

4.2.2 A fill-model ladder: from toy assumptions to research-grade simulators

It is useful to structure fill simulation as a ladder of increasing realism. Each rung adds a specific capability and requires specific datasets.

Rung	Data needed	What it gets right / wrong
(1) Touch-fill	L2 top-of-book (or mid/BBO)	Right: simple spread capture intuition. Wrong: massively overstates passive fills; ignores priority and partials.
(2) Trade-through fill	Trades + L2	Right: fills require traded volume to pass your price. Wrong: still optimistic without modeling volume ahead and cancellations.
(3) Probabilistic queue-based fill	L2 deltas + trades	Right: fill likelihood depends on volume ahead and event flow; partial fills possible. Wrong: still a reduced-form approximation of the matching engine.
(4) Event-driven queue simulation	L2 deltas (and cancellation proxies) + calibration	Right: explicit queue dynamics; can validate against stylized facts. Wrong: needs careful fitting and ongoing re-estimation.

Table 4.2: A practical progression of fill models for passive execution backtests.

Two research facts should shape how you choose a rung:

- Avellaneda–Stoikov class market-making models tie profitability to quoting distance and fill intensities that must be estimated and

can drift. If your fill model cannot capture drift, your “optimal” spread and inventory behavior will be brittle.

- Priority in the queue is a first-order variable for passive execution; ignoring it can materially bias simulated fill rates and expected PnL for market-making strategies.

4.2.3 Latency modeling: turning market data into “as-observed” state

Even with perfect historical orderbook events, your strategy did not observe them instantly in production. A latency model is the bridge between the event tape and the strategy’s decision state.

4.2.4 Three clocks and two latencies

Maintain two explicit delays:

- **Market-data latency** (exchange event → your strategy sees it).
- **Order-action latency** (your decision → exchange acknowledges and matches it).

In practice you represent these as random variables with distributions (or at least fixed quantiles) estimated from production logs. The simulator then:

- replays market events into a *latency buffer*,
- exposes to the strategy only those events whose release time has arrived,
- routes outbound orders through an *action buffer* before they affect the simulated matching process.

4.2.5 Why latency creates non-deterministic fills

Once you include latency, two identical strategy runs can diverge if:

- event ordering at the same timestamp is ambiguous,
- two messages arrive in different micro-order due to jitter,
- a cancel races a fill (sometimes the cancel “wins,” sometimes it does not).

A credible simulator therefore needs a reproducible randomness source (seeded RNG) and must log the realized latency draws so you can replay a specific run.

4.2.6 Queue and priority approximations without a full matching engine

With public L2 data you rarely know the true FIFO queue at each price level, but you can still model priority effects in a way that (i) uses observable state and (ii) penalizes overly optimistic fills.

4.2.7 State variables that matter

As we saw when discussing queue state variables earlier, the key quantities for a passive order at price level p are:

- $Q_t(p)$: total resting size at p on your side (observable from L2 depth),
- q_t : your estimated size ahead of you (latent, modeled),
- the flow of **marketable trades** that consume liquidity at p and better,
- the flow of **cancellations** at p (partially observable through L2 reductions).

Queue-reactive/queueing models treat these as state-dependent processes: event intensities (trades, cancellations, new limit orders) depend on the current queue sizes. This is a more realistic basis for passive fill simulation than distance-only fill intensities.

4.2.8 A robust reduced-form model for passive fills

A practical approach that works well in research pipelines is:

- When you place a passive order of size s at price p , initialize an *ahead estimate*:

$$q_{t_0} \leftarrow \alpha Q_{t_0}(p),$$

where $\alpha \in [0, 1]$ captures how late you arrived to the level. You calibrate α from live fills or use conservative defaults (larger α means more volume ahead, fewer simulated fills).

- Update q_t as L2 deltas arrive. If displayed size at p increases, assume it joins behind you (do not increase q_t). If displayed size at p decreases, attribute some fraction to cancellations ahead (reducing q_t) and the rest to executions (also reducing q_t), using trade prints to inform the split.
- When cumulative traded volume through p (accounting for better prices) exceeds q_t , you begin filling. Partial fills occur naturally when trade-through volume arrives in bursts.

This model is intentionally conservative if you attribute fewer reductions to cancellations ahead of you and assume some adverse selection (e.g., trades accelerate when your quote is most exposed).

4.2.9 Order lifecycle in simulation: placement, partial fills, and cancellation races

A high-fidelity simulator is event-driven: both the market and your strategy generate events, and the engine resolves them in time order subject to latency buffers.

4.2.10 Sequence of events you must represent

At minimum, model the following transitions:

- **Submit**: strategy emits an order intent.

- **Arrive:** after action latency, the order becomes active in the simulated book (or joins a virtual queue state).
- **Fill:** one or more partial fills occur as marketable flow consumes liquidity.
- **Cancel/modify:** strategy attempts to remove or change the order; the request arrives after latency.
- **Race resolution:** if a fill and cancel overlap, resolve probabilistically or with a deterministic rule tied to timestamps and a jitter model.

Notice that nothing here requires you to reproduce the venue's private matching rules exactly; it requires you to capture *the economic consequences* (missed cancels, partials, and delayed actions).

4.2.11 Sequence diagram: simulated order lifecycle with latency and probabilistic fills

This subsection denotes the conceptual sequence diagram for order lifecycle, with market events, latency buffers, and probabilistic race resolution driving fills and cancellations.

4.2.12 Partial fills and execution price formation

Partial fills should not be treated as a bookkeeping detail; they change the strategy's realized risk:

- Inventory accumulates gradually, affecting subsequent quotes and hedges.
- Fees and maker rebates (if any) apply per fill, not per order intent.
- The average execution price depends on the time profile of fills, not just the final filled size.

In simulation, store a fill as a stream of *fill lots*:

$$\{(t_i, \Delta s_i, p_i)\}_{i=1}^n,$$

so you can compute realized PnL, funding exposure windows, and inventory paths correctly. This also makes it possible to match your simulator output to live “user fills” records later.

4.2.13 Non-determinism: embrace it, but make it reproducible

In real markets, two runs are never identical. Your simulator should include randomness in:

- latency draws (both market-data and action),
- race resolution when timestamps collide or are within a jitter band,
- probabilistic fills when only partial state is observable.

But reproducibility is mandatory for research. For each backtest run:

- log the RNG seed,
- log the realized latency samples,
- log every race resolution (which side won and why).

This converts “the backtest changed” into a debuggable artifact.

4.2.14 Calibration and drift: linking fill intensities to market state

Avellaneda–Stoikov style quoting depends on fill intensities that are functions of quote distance and that drift over time. In practice, your simulator needs at least two layers of calibration:

- **Static baseline:** estimate how fill probability (or intensity) decays as your quote moves away from the touch.

- **State dependence:** condition those estimates on a small set of regime-relevant variables (e.g., spread, short-horizon volatility, and orderbook imbalance).

Even a simple “two-regime” calibration (calm vs fast) can materially improve realism because it prevents the simulator from applying calm-market fill rates during breakouts.

A practical fitting target is the conditional probability of a fill within a horizon Δt :

$$\mathbb{P}(\text{fill within } \Delta t \mid d, \text{state}),$$

where d is your distance from the best price. This can be estimated from historical L2/trade data by labeling periods where a hypothetical passive order would have been filled under your queue approximation. Re-estimate on a rolling window to track drift.

4.2.15 Unit tests for a book builder and fill simulator

Before you debate model sophistication, ensure basic invariants hold. These tests should run on every dataset shard and every code change.

4.2.16 Orderbook reconstruction tests

Given “apply snapshot then deltas”:

- **No crossed book:** best bid $<$ best ask at all times (except during isolated message gaps, which should be detected and flagged).
- **Non-negative depth:** no level has negative size after applying deltas.
- **Monotone sequencing:** detect any sequence gaps; halt replay or force a resync snapshot boundary.

4.2.17 Execution-simulator tests

- **Conservation:** total filled size for an order never exceeds its original size.
- **Price consistency:** passive fills occur at your limit price (or better, if your model allows price improvement), not at mid.
- **Race handling:** when a cancel and a fill overlap, the outcome is consistent with your latency model and logged for replay.

4.2.18 A practical architecture: event-driven replay + modular execution models

A robust implementation splits into three modules:

- **Market replay:** rebuilds the as-observed book from snapshots/deltas and emits a time-ordered event stream.
- **Latency layer:** delays market events to the strategy and delays strategy actions to the engine.
- **Execution model:** converts market events into fills for active orders (from trade-through rules up to queue-reactive simulation).

This modularity matters because you can upgrade the execution model (e.g., from rung 2 to rung 3 in the fill-ladder table) without rewriting your market-data and latency plumbing.

4.2.19 When to stop adding realism

The right simulator is the simplest one that (i) matches key live diagnostics and (ii) fails conservatively when uncertain. If you find yourself adding complexity that cannot be calibrated from your stored data (or validated against stylized facts), you are likely fitting noise.

The discipline is:

- start with a conservative baseline (trade-through + latency),

- add priority-aware partial fills,
- only then move to queue-reactive dynamics, and validate at each step.

That workflow produces backtests that are not only more believable, but also more actionable: when performance changes, you can attribute it to a specific modeling assumption rather than a black-box simulator.

4.3 The Strategy Research and Iteration Loop

The previous sections established two pillars: (i) you need market data that can be replayed deterministically from raw events, and (ii) you need a fill simulator that penalizes unrealistic execution assumptions (latency, partial fills, and priority effects). This section adds the **third** pillar: a disciplined research loop that prevents overfitting and provides a controlled path from idea to production.

The central mindset shift is that a trading strategy is not a single backtest. It is a **sequence of experiments** whose inputs and outputs must be versioned, audited, and comparable. Without that structure, iterative improvements are indistinguishable from curve-fitting.

4.3.1 The research loop as an engineering system

A robust workflow separates the work into stages with explicit artifacts. A practical template is:

Hypothesis → Dataset build → Backtest → Attribution review → Parameter update → Paper trading → Live rollout.

Each arrow is a handoff between modules, and each module produces outputs you should be able to reproduce months later.

4.3.2 Why this loop matters specifically on on-chain perps

On-chain perp markets are fast enough to demand microstructure realism, but they also have operational quirks (connectivity, dropped messages, heterogeneous data sources) that can silently corrupt conclusions. Two practical consequences follow directly from the research notes:

- Robust research loops require separating **streaming (low-latency)** from **batch/backfill**, with explicit **data provenance** (public API vs node vs vendor) and caching to handle rate limits.
- Simulation credibility improves when simulator parameters and state representations are **fit to real data and continuously re-fit**, rather than being fixed once.

4.3.3 Stage 1: Hypothesis and falsifiable predictions

A hypothesis should be stated in a form that can fail. For orderbook strategies, good hypotheses are typically conditional statements about microstructure:

- “When spread widens and imbalance is extreme, passive quotes at d ticks have higher fill probability but worse post-fill drift.”
- “Funding persistence predicts short-horizon mean reversion in the premium, but only when open interest growth is flat.”

Bad hypotheses are vague (“market making is profitable”) or post-hoc (defined after scanning results).

Define the decision timing. Before any backtest, specify exactly when the strategy is allowed to observe data and act. If you allow a signal computed with exchange-time alignment but execute with receive-time latency, you can leak information without noticing. Decide whether the experiment is:

- **as-observed** (closest to live conditions), or
- **exchange-time** (useful for structural estimation, but must be guarded against look-ahead).

4.3.4 Stage 2: Dataset build (streaming vs batch/backfill)

High-quality research starts with a dataset that is both complete and well-labeled.

4.3.5 Streaming capture (for correctness)

Streaming data is the canonical record of what happened and when you learned it. The streaming system should store:

- raw immutable event logs (snapshots/deltas/trades),
- ingestion sequence indices and timestamps,
- integrity flags (e.g., any evidence of message drops or resync events).

This is the dataset you trust when diagnosing discrepancies between backtest and live.

4.3.6 Batch/backfill (for coverage and long windows)

Batch methods are how you obtain long history, re-download missing intervals, and build research tables efficiently. The operational requirement is **provenance**: label each segment by its source (public API vs node vs vendor), retrieval time, and any transformations applied. This is not bureaucracy; it is how you detect that a new provider normalized fields differently or omitted certain events.

Caching and rate limits. Batch queries should be cached with content-addressed keys (e.g., request parameters plus endpoint ver-

sion). Rate limits are not just an uptime nuisance; they create holes in history that show up as “mysterious” regime changes.

4.3.7 Schema constraints are part of the dataset

Exchange constraints and microstructure parameters (size decimals, leverage constraints, isolated margin flags) should be first-class inputs in research code to avoid infeasible backtest trades. In practice:

- store the per-asset constraints from `metaAndAssetCtxs` alongside market data,
- validate every simulated order against these constraints at the time it would have been sent,
- record validation failures as events (do not silently clip sizes or round in a way that changes strategy intent).

4.3.8 Stage 3: Backtest design that resists overfitting

Overfitting in HFT often comes from two sources: (i) too many degrees of freedom in signals/parameters, and (ii) execution assumptions that inadvertently add free lunch. You address both with design choices.

4.3.9 Use strict evaluation windows (walk-forward)

A standard structure is:

- **train:** fit parameters and calibrations (including fill model parameters),
- **validation:** select between competing variants,
- **test:** one-time final evaluation (do not iterate on it).

For non-stationary markets, use walk-forward evaluation: re-fit on a rolling window and test on the next slice. This matches the reality that simulator parameters and state representations must be continuously re-fit.

4.3.10 Keep the fill model fixed while testing signal changes

A common failure mode is changing both alpha logic and execution assumptions simultaneously. To isolate causality:

- treat the fill simulator as a **shared infrastructure component**,
- upgrade it on its own schedule, and
- re-run a stable benchmark strategy whenever you change the simulator.

This prevents you from attributing improved results to a new signal when the real driver is a more optimistic fill rule.

4.3.11 Prefer small, interpretable parameter sets

In microstructure strategies, a few parameters often dominate:

- quote distance / skew,
- inventory aversion,
- cancellation aggressiveness,
- participation limits under volatility.

If an improvement requires dozens of knobs to work, it is usually fragile.

4.3.12 Stage 4: Attribution review as the gatekeeper

Raw PnL is not the decision criterion during research. The criterion is: **do the drivers make sense, and are they stable?**

Before promoting a change, require an attribution review that answers:

- Did returns come from **spread capture** or from **mark-to-market drift**?
- How much performance depends on **funding** versus trading edge?
- Are gains offset by **slippage**, missed cancels, or inventory blowups in fast markets?
- Are improvements concentrated in a narrow subset of days or regimes?

This is where replayability pays off: you can drill into specific intervals and re-simulate the exact same event tape under alternative parameterizations.

4.3.13 Stage 5: Experiment tracking (make iteration cumulative)

If you cannot reproduce a result, it is not research; it is a story. Implement an experiment ledger that records, at minimum:

- **dataset version hash**: identifies the exact raw logs and derived tables,
- **simulator version**: code commit or semantic version for the fill/latency model,
- **parameter set**: all strategy and simulator parameters (including calibration window lengths),
- **evaluation windows**: start/end timestamps for train/validation/test and walk-forward steps,

- **feature definitions:** code version for any engineered features.

This checklist is especially important when you maintain both streaming-captured datasets and batch/backfilled datasets: it is easy to accidentally “improve” results by switching sources.

4.3.14 Stage 6: Paper trading as a simulator audit

Paper trading is not only for de-risking capital. It is your most direct test of simulator realism.

4.3.15 Reality-gap diagnostics

Use paper trading to compare simulated vs realized behavior along dimensions your fill model claims to capture:

- **Fill rates by bucket:** spread, imbalance, and volatility buckets; if the simulator overstates fills in wide-spread regimes, your quoting model will look too good.
- **Inventory distribution:** compare the time spent at different inventory levels; mis-modeled partial fills often show up as unrealistic inventory variance.
- **Implementation shortfall (IS):** compare realized vs expected IS; persistent differences are a sign your latency or priority approximation is biased.

The outcome should feed back into calibration: simulation credibility improves when parameters are fit to real data and continuously re-fit.

4.3.16 A/B changes with incremental rollout

When you change strategy logic, avoid full switches. Use A/B changes:

- run baseline and candidate variants on the same market in parallel (paper first),

- keep risk budgets small and symmetric,
- compare attribution components, not just headline PnL.

Incremental rollout means moving from paper to small live size, then scaling only after the same diagnostics remain stable.

4.3.17 Stage 7: Production promotion criteria and monitoring for decay

Promotion should be gated by both statistical and operational criteria.

4.3.18 Promotion checklist

Require:

- out-of-sample performance that persists across multiple market conditions,
- stable attribution (no hidden dependency on a single regime),
- paper-trading reality-gap metrics within tolerance,
- feasibility under exchange constraints (size decimals, leverage constraints, isolated-only flags).

4.3.19 Monitoring for structural change

Even good strategies decay. In production, continuously track:

- drift in fill rates at fixed quote distances,
- changes in spread/volatility distribution,
- changes in the relationship between imbalance and short-horizon returns.

When these drift, you do not immediately “optimize” parameters. First, you re-run the research loop: rebuild the dataset slice, re-fit simulator parameters, and re-evaluate variants under walk-forward.

4.3.20 Putting it together: a workflow that scales

The practical outcome of this methodology is speed **without** self-deception:

- Streaming capture gives you a canonical, replayable event tape.
- Batch/backfill gives you breadth and long history, with provenance and caching.
- The simulator enforces execution realism (partial fills, latency, and priority effects).
- Experiment tracking and attribution reviews ensure that improvements are real and repeatable.
- Paper trading and incremental rollout close the loop by measuring the reality gap directly.

Run this loop consistently and your strategy development becomes cumulative: each iteration either strengthens the system or produces a clear failure that teaches you what to fix next.

4.4 Regime Detection and Volatility Filters

Orderbook strategies are *context dependent*. The same quoting logic that prints steady spread capture in a calm, liquid market can lose quickly when volatility jumps, spreads gap, or aggressive flow becomes one-sided. The prior sections give you the tools to (i) reconstruct the market state faithfully and (ii) simulate execution under latency and priority effects. This section builds on that foundation by adding a regime layer: a set of online signals that classify the environment and *adapt* your strategy's aggressiveness in real time.

The objective is not to forecast direction. It is to avoid applying the wrong behavior to the wrong market.

4.4.1 What a “regime” means for an orderbook strategy

For systematic market making and other orderbook-driven strategies, a regime is a configuration of microstructure and risk conditions that changes:

- **Fill rates** at given quote distances (how quickly passive orders get hit),
- **Adverse selection risk** (how often fills are followed by unfavorable mid moves),
- **Transaction costs** (spread, fees, and implied impact),
- **Inventory risk** (how quickly inventory can accumulate and how expensive it is to unwind).

A useful regime classifier does two things:

- summarizes the environment with a small, tractable state representation (recent research emphasizes variables such as spread and volume imbalance),
- converts that summary into explicit controls: quote width, size, participation, leverage, or an on/off switch.

4.4.2 Feature design: a small state vector that is hard to fool

Regime detection is only as good as its inputs. You want features that are:

- **observable in real time** (no look-ahead),
- **stable across assets** (to reuse models),
- **linked to execution** (so the filter improves trading, not just classification accuracy).

The research notes suggest a practical feature set spanning candles, the orderbook, and /info context.

4.4.3 Volatility: candle-based realized volatility

Use rolling realized volatility from 1m/5m candles (or equivalent higher-frequency returns if you compute them from mid prices). Example definitions:

$$\sigma_t^{(1m)} = \sqrt{\sum_{i=1}^N r_{t-i}^2}, \quad r_t = \ln \left(\frac{P_t}{P_{t-1}} \right),$$

where P_t is mid or mark sampled at a consistent cadence.

Why candles are still useful here: volatility is fundamentally a price-path property, and candles provide a robust, low-noise summarization that is easy to compute online.

4.4.4 Liquidity and microstructure: spread, depth, and imbalance

At minimum, include:

- **Top-of-book spread** (absolute or in bps),
- **Displayed depth** at the best bid/ask (or within the first k levels),
- **L2 imbalance:**

$$I_t = \frac{D_{bid,t} - D_{ask,t}}{D_{bid,t} + D_{ask,t}},$$

where $D_{bid,t}$ and $D_{ask,t}$ are depth measures from your reconstructed L2 book.

These align with the “tractable state representation” principle: spread and imbalance are compact proxies for trading conditions that also plug directly into fill and impact modeling.

4.4.5 Flow intensity: trade arrival rate spikes

Compute a rolling trade arrival rate or traded notional per second:

$$\lambda_t = \frac{N_{trades}}{\Delta},$$

where N_{trades} is the number of trades in $(t - \Delta, t]$. Spikes in λ_t often correspond to information events, liquidation cascades, or broad market repricing. For passive strategies, those are precisely the times when latency and priority effects become most punitive.

4.4.6 Context features from `metaAndAssetCtxs`: **impact, funding, premium, open interest**

From `metaAndAssetCtxs` asset contexts, features that tend to be regime-informative include:

- `impactPxs` widening (a coarse execution-cost proxy),
- funding and its persistence (carry pressure and crowding),
- premium (mark vs oracle divergence proxy),
- `openInterest` changes (positioning build-up),
- `dayNtlVlm` as a baseline liquidity scale.

These are especially useful when the visible orderbook does not fully reflect effective liquidity (e.g., when displayed depth is thin but impact is stable, or vice versa).

4.4.7 Two layers of filters: **risk scaling and hard stops**

Regime tools work best as a two-layer system:

- **Continuous risk scaling:** smoothly adjust quote width, size, and leverage.
- **Discrete safety stops:** turn the strategy off when conditions exceed a hard limit.

This avoids an all-or-nothing posture while still protecting the bot from tail events.

4.4.8 Volatility scaling

A simple, robust rule is to scale aggressiveness inversely with realized volatility:

$$w_t = w_0 \cdot \left(\frac{\sigma_t}{\sigma_{ref}} \right), \quad s_t = s_0 \cdot \left(\frac{\sigma_{ref}}{\sigma_t} \right),$$

where w_t is quote half-spread (or an additive widening), and s_t is order size. This captures the core intuition: when price moves faster, you demand more edge per trade and take smaller inventory increments.

4.4.9 Liquidity scaling using spread and impact proxies

Volatility alone is not enough: sometimes price is calm but liquidity is thin (wide spreads, high impactPxs). Add explicit widening or size reductions based on transaction-cost proxies:

$$w_t \leftarrow w_t + \beta \cdot spread_t,$$

and if impactPxs indicate higher cost for the size you plan to trade, reduce size or back off to deeper levels. This connects regime detection to actual implementation costs.

4.4.10 Hard stops (algorithmic on/off switches)

Define hard constraints that disable quoting when the probability of paying a large execution penalty is too high. Common triggers:

- spread exceeds a threshold (absolute or bps),
- impact proxy exceeds a threshold (e.g., impactPxs diverge sharply from mid),
- trade intensity exceeds a threshold (“fast market”),
- volatility exceeds a threshold,
- internal health: data integrity alarms (e.g., suspected message gaps) or latency spike.

The last item matters: a strategy should not trade microstructure signals if your replay state is compromised.

4.4.11 Online changepoint detection (BOCPD) as a regime engine

Thresholds and scaling rules are useful, but they do not explicitly answer: *did the regime change?* Online changepoint detection is designed for that question.

4.4.12 What BOCPD provides

Online changepoint detection (BOCPD) provides a principled streaming method to infer regime shifts; it outputs a **run-length distribution** (time since last change) that can drive risk scaling and recalibration triggers.

Intuition: BOCPD maintains probabilities over how long the current regime has lasted. When the model sees data that is unlikely under the current regime, probability mass shifts toward short run lengths (“a fresh change likely occurred”).

4.4.13 What to feed into BOCPD

BOCPD is not limited to prices. In microstructure trading, it is often more effective to run BOCPD on quantities that directly affect execution:

- realized volatility estimates,
- spread series,
- fill-rate proxies (from simulator diagnostics or paper trading),
- short-horizon adverse selection proxies (e.g., post-fill mid move estimates),
- trade arrival rates.

This reflects a key point from the simulation research notes: tractable state representations tie regime detection to fill/impact modeling.

4.4.14 Using the run-length distribution for controls

Two practical control patterns:

- **De-risk on fresh changes:** if run-length probability mass concentrates near zero (high probability of a new regime), widen quotes, cut size, and reduce leverage temporarily. This is a “cooldown” period while the model gathers evidence.
- **Trigger re-fitting:** if BOCPD flags a changepoint in fill-rate or adverse-selection proxies, schedule re-estimation of simulator parameters (fill intensity vs distance, state-conditional fill probabilities). This is directly aligned with the research note that simulator parameters should be continuously re-fit, not frozen.

4.4.15 Connecting regimes to quoting parameters (a control map)

A regime classifier is only useful if it maps cleanly to strategy knobs. A practical control map for a market-making style bot might be:

- **Quote width:** increase with volatility, spread, and “fresh regime” probability.
- **Order size:** decrease with volatility, impact proxy widening, and fast trade intensity.
- **Refresh/cancel aggressiveness:** increase in fast markets (but tempered by action latency; overly frequent cancels can create race losses).
- **Inventory limits:** tighten in trending/high-vol regimes; loosen modestly in stable mean-reverting regimes.
- **Leverage:** scale down as volatility and impact costs rise; enforce hard caps when changepoints occur.

- **Participation rate:** cap the fraction of observed traded volume you try to capture (lower in fast/high-impact regimes).

This control map is intentionally interpretable. If a filter changes behavior and performance shifts, you can explain why.

4.4.16 Avoiding a common trap: regime filters that overfit

Regime models can overfit just like alpha signals. Three safeguards help:

4.4.17 Guardrail 1: Evaluate filters by stability, not only PnL

A good filter:

- reduces left-tail outcomes (drawdowns) more than it reduces mean returns,
- stabilizes inventory distribution and execution costs,
- improves robustness across walk-forward windows.

4.4.18 Guardrail 2: Keep regimes coarse

Two to four regimes are often enough for control purposes (calm, normal, fast, stressed). If you need ten micro-regimes to see an improvement, you are likely fitting noise.

4.4.19 Guardrail 3: Validate with reality-gap diagnostics

As discussed in the iteration loop, compare simulated vs realized fill rates and execution costs by regime bucket. If the simulator systematically overstates fills in “fast” regimes, the filter may look unnecessary

in backtests and essential in live trading—a mismatch you can detect only through consistent diagnostics.

4.4.20 A practical implementation order

To integrate regime detection without destabilizing your system:

- Start with volatility scaling + spread/impact hard stops (simple, effective, easy to debug).
- Add trade intensity and imbalance as additional scaling inputs.
- Introduce BOCPD for changepoint-aware cooldowns and to schedule parameter re-fitting.
- Finally, calibrate state-conditional fill models and quote parameters per regime.

This progression keeps the logic grounded: each layer is motivated by an observable change in execution risk, not by a desire to add complexity.

4.5 Performance Attribution

A backtest (even a good one) outputs a single number: PnL. But PnL alone does not tell you *why* a strategy made or lost money, nor whether the mechanism is stable. For on-chain perps, that “why” is particularly important because returns are a mixture of:

- **structural alpha** from your trading logic,
- **funding** (carry) that can dominate PnL and flip signs across regimes,
- **fees or maker rebates** that can turn a marginal edge into a losing strategy,
- **execution slippage and opportunity costs** caused by latency, partial fills, and non-execution.

This section provides a microstructure-aware attribution framework that is compatible with the data and simulation tooling built earlier in the chapter, and that supports live reconciliation using Hyperliquid user-level records (fills, funding payments, and account state with positions/uPnL).

4.5.1 Attribution: what it is and what it is not

Portfolio attribution frameworks (CFA review; Brinson lineage) provide useful terminology (allocation vs selection, contribution, benchmarks). However, trading attribution must add microstructure components: slippage/impact, missed fills, latency, and other execution frictions. Otherwise you risk calling “alpha” what is actually fee arbitrage, or calling “slippage” what is actually adverse selection.

A practical definition for systematic trading is:

Performance attribution is the decomposition of realized PnL into components that correspond to distinct economic mechanisms, each of which can be measured, monitored, and optimized separately.

4.5.2 The minimal attribution tree for on-chain perps

Start with an attribution tree that you can compute for any day and any strategy variant. A robust baseline tree is:

1. **Gross trading PnL** (from fills)
2. **Fees / maker rebates**
3. **Funding payments**
4. **Inventory mark-to-market** (unrealized PnL due to holding risk)
5. **Execution shortfall vs decision price** (implementation shortfall)

6. **Residual / model error** (anything not explained above)

Two points from the research notes are decisive here:

- Funding history is a distinct PnL driver in perps and must be separated from trading alpha; data-access constraints (rate limits/availability) affect how precisely funding PnL can be reconstructed historically.
- Hyperliquid user-level data streams/queries (fills, funding payments, account state with positions/uPnL) are required to reconcile live PnL with model PnL and to build an attribution tree that includes fees/funding and inventory mark-to-market.

4.5.3 Compute the basic PnL components

4.5.4 1) Trading PnL from fills

Let fills be indexed by i , each with signed quantity q_i (positive for buy, negative for sell), execution price p_i , and timestamp t_i . For a given instrument, define the signed cashflow:

$$CF = - \sum_i q_i p_i,$$

and the position after fills:

$$Q = \sum_i q_i.$$

To mark the position, choose a consistent mark price m_t (e.g., mark or mid at evaluation time). Then:

$$TradingPnL = CF + Q \cdot m_T.$$

This is the simplest fill-based decomposition. For intraday attribution you compute this cumulatively and separate realized vs unrealized via position changes.

4.5.5 2) Fees and maker rebates

Fees (or rebates) attach directly to fills. Represent them as:

$$Fees = \sum_i f_i,$$

where f_i is signed so that fees are negative contributions and rebates are positive. The key is to store the fee fields as part of your fill records so you can compute this without inference.

4.5.6 3) Funding PnL

Funding is carry; it is not “trading edge.” Compute it from user funding payment events if available; if you must reconstruct historically, use funding history and positions, while acknowledging potential sampling/coarsening.

At a high level:

$$FundingPnL = \sum_k \left(-Q_k \cdot m_k \cdot r_k^{fund} \right) \Delta t_k,$$

where Q_k is your position over interval k , m_k is an appropriate mark (often the venue mark), and r_k^{fund} is the funding rate over that interval. The precise formula depends on the venue’s funding convention; the attribution principle is to isolate this term from trading PnL so that a strategy that “wins” only by collecting carry is not mistaken for a microstructure alpha.

4.5.7 4) Inventory mark-to-market

Even if your strategy aims to be neutral, inventory is usually nonzero. The mark-to-market component answers: did you make money because you *held* risk, not because you *traded well*?

A clean way to isolate this is to compute PnL at a high sampling rate and separate:

- fill-driven changes (realized),

- mark-driven changes on held inventory (unrealized drift).

This component is essential for diagnosing when a “market making” strategy is actually running an implicit trend-following exposure due to inventory accumulation.

4.5.8 Implementation shortfall (IS): the execution-centric benchmark

Implementation shortfall (Perold, 1988) benchmarks execution versus the decision price and decomposes performance shortfall into execution cost and opportunity cost—particularly relevant for limit-order strategies where non-execution is material.

4.5.9 Decision price and executed price

Suppose at time t_0 your strategy decides to buy target quantity $Q^* > 0$. Let:

- p_0 be the *decision price* (commonly the mid at t_0 , using the as-observed book),
- Q^e be the executed quantity within the allowed horizon,
- $\bar{p}^e$ be the volume-weighted average execution price (VWAP of your fills),
- p_T be the terminal benchmark price at the end of the horizon.

4.5.10 Total implementation shortfall

For a buy, the total shortfall (in quote currency) can be expressed as:

$$IS = Q^e(\bar{p}^e - p_0) + (Q^* - Q^e)(p_T - p_0).$$

For a sell, the signs flip accordingly. The interpretation:

- The first term is **execution cost**: how much worse your fills were than the decision price.

- The second term is **opportunity cost**: what it cost you to *not* get filled when the market moved away.

This decomposition is especially useful for passive strategies:

- If you widen quotes too much, execution cost may look great (you only fill on favorable terms) but opportunity cost explodes (you miss the trade).
- If you quote too tight, you fill often but may pay via adverse selection and negative post-fill drift (execution cost and subsequent mark-to-market losses).

4.5.11 Microstructure-aware IS for continuous quoting

A quoting strategy has many “decisions” per day. Treat each quote placement (or refresh) as a decision, with a decision price defined by the contemporaneous mid. Aggregate IS over:

- order lifetimes (place-to-cancel windows),
- fixed time buckets (e.g., 1 minute),
- regime buckets (spread/volatility/imbalance states from the previous section).

This is where your backtesting infrastructure matters: you need consistent as-observed timestamps and a fill simulator that generates partial fills and cancel races, otherwise IS will be systematically biased.

4.5.12 Reconciling backtest attribution with live attribution

Attribution is most valuable when it helps you close the gap between what the model predicted and what live trading realized.

4.5.13 The reconciliation dataset

To reconcile, you need three categories of records:

1. **Model-side:** your backtest's predicted fills, fees, funding accrual, inventory path, and IS.
2. **User-side:** Hyperliquid user-level records: fills, funding payments, and account state with positions/uPnL.
3. **Market-side:** the public event tape (L2 + trades + funding history) used to reconstruct the state and define decision prices.

Do not try to reconcile using only market data; without user fills and funding payments you cannot uniquely identify fee/funding effects and realized execution.

4.5.14 A practical discrepancy checklist

When live PnL deviates from model PnL, triage in this order:

1. **Data integrity:** confirm no message-drop intervals tainted the replay state.
2. **Fills:** compare expected fills vs actual user fills; difference here usually indicates priority/latency bias in the simulator.
3. **Fees:** verify fee schedules and maker/taker classification in the fill records.
4. **Funding:** reconcile funding payments against the position path and funding history; note that data-access constraints can reduce historical precision.
5. **Marking:** ensure consistent mark/mid usage for unrealized PnL and IS benchmarks.

This checklist mirrors the logic of the attribution tree: you localize the error to a specific mechanism rather than “the strategy changed.”

4.5.15 A cumulative PnL breakdown (example chart)

The most effective way to communicate attribution is to plot cumulative PnL and stack (or waterfall) its components so that “edge” and “friction” are visually separated.

4.5.16 Using attribution to improve the strategy (not just explain it)

Attribution is diagnostic only if it changes what you do next. Common actions tied to specific attribution failures:

- **Slippage dominates:** widen quotes, reduce participation in fast regimes, improve latency assumptions, or tighten hard stops.
- **Fees overwhelm edge:** ensure you are earning maker rebates when intended, reduce churn, and avoid excessive quote refresh that creates taker-like behavior.
- **Funding dominates PnL:** explicitly decide whether you are running a carry strategy; if not, neutralize funding exposure or separate it into a distinct sleeve.
- **Inventory mark-to-market dominates:** tighten inventory limits, add skew, or introduce regime-based de-risking to prevent implicit directional bets.
- **Opportunity cost dominates IS:** your quotes are too conservative or your cancels are too aggressive; revise size/width trade-offs or the cancel policy under latency.

Chapter 5

Systematic Trading and Market Making Strategies

From harvesting asymmetric funding rates to providing continuous two-way liquidity on an on-chain orderbook, systematic trading on Hyperliquid demands a unique blend of quantitative rigor and DeFi-native execution. This chapter bridges the gap between theoretical models and live capital deployment. We will dissect the mechanics of directional alpha, delta-neutral basis strategies, and the intricate calculus of market making—complete with inventory control, adverse selection management, and the role of native HLP vaults. Whether you are hunting for millisecond arbitrage opportunities or seeking to engineer robust, continuous quoting algorithms, these frameworks will serve as your blueprint for thriving in decentralized derivatives markets.

5.1 Directional Strategies

5.1.1 A funding-aware objective: edge must beat carry & friction

A directional perp trade has three main components over a holding window $[t, t + H]$:

1. **Price move:** $\Delta p = p_{t+H} - p_t$
2. **Funding cashflows:** hourly transfers based on position notional (funding settles hourly on Hyperliquid)
3. **Execution costs:** spread/impact plus fees (maker rebates or taker fees)

Write the position in contracts (or base units) as q_t (positive for long, negative for short). Let P_u^{oracle} be the oracle price used for funding notional conversion (per Hyperliquid docs), and let f_u be the hourly funding rate (a signed rate; positive means longs pay shorts). If we hold q constant over hours $u = t, t+1, \dots, t+H-1$, the *expected* contribution from funding is approximately

$$\mathbb{E}[\text{FundingPnL}] \approx -q \sum_{u=t}^{t+H-1} \mathbb{E}\left[f_u P_u^{\text{oracle}}\right].$$

For intuition:

- If you are **long** ($q > 0$) and funding is expected **positive** ($f_u > 0$), the term is negative: funding is a drag.
- If you are **short** ($q < 0$) and funding is expected **positive**, the term is positive: you expect to earn funding.

Your signal provides an expected price change $\mathbb{E}[\Delta p]$ over horizon H . Ignoring convexities, expected trade PnL becomes

$$\mathbb{E}[\text{PnL}] \approx q \mathbb{E}[\Delta p] - q \sum_{u=t}^{t+H-1} \mathbb{E}[f_u P_u^{\text{oracle}}] - \mathbb{E}[\text{Costs}].$$

the funding-aware expected value is the core translation step: a “good” momentum or mean-reversion forecast is *not* enough; it must also overcome the expected funding drag (if any) and the friction of getting in and out.

5.1.2 Practical funding expectation

Hyperliquid funding is computed from a premium index (impact prices vs oracle) plus a clamped interest-rate component, sampled frequently and settled hourly. You typically cannot forecast that full mechanism precisely, but you can build a workable expectation with:

- **Current funding rate as baseline:** $\hat{f}_{t:t+H-1} = f_t$ (a myopic but stable assumption).
- **Mean-reverting funding model:** $\hat{f}_{t+k} = \bar{f} + (f_t - \bar{f})\rho^k$ with $\rho \in (0, 1)$.
- **Regime filter:** treat large absolute funding as a congestion signal; cap your holding time when $|f_t|$ is large.

A minimal, robust approximation is:

$$\widehat{\text{FundingCost}}(q, H) = -q \cdot P_t^{\text{oracle}} \cdot H \cdot \hat{f}_t,$$

where H is measured in hours. This is not perfect (oracle price moves, funding changes), but it forces the right behavior: you stop holding longs when long funding is expensive unless your forecast is strong enough.

5.1.3 From signal to trade: a funding-aware decision rule

Directional systems are easiest to implement cleanly by separating *forecasting* from *decisioning*. The forecasting block outputs a signed expected return (or z-score). The decision block applies funding, costs, and risk constraints.

5.1.4 A general decision inequality

Let $\hat{r}_{t,H}$ be the expected return over H (e.g., $\hat{r} = \mathbb{E}[\Delta p]/p_t$), and let $\hat{c}_{t,H}$ be expected total costs in return units (fees + spread/impact for entry and exit). Let $\hat{f}_{t,H}$ be expected cumulative funding over H (signed; in return units). For a *long*:

$$\text{Enter/hold long if } \hat{r}_{t,H} - \hat{f}_{t,H} - \hat{c}_{t,H} > 0.$$

For a *short*, the sign flips in the intuitive way:

$$\text{Enter/hold short if } -\hat{r}_{t,H} + \hat{f}_{t,H} - \hat{c}_{t,H} > 0.$$

These inequalities turn funding into an explicit *threshold* that suppresses marginal trades. In practice, you add a margin for model error:

$$\hat{r}_{t,H} - \hat{f}_{t,H} - \hat{c}_{t,H} > \theta,$$

where $\theta > 0$ increases when volatility rises, when liquidity is thin, or when your model is known to degrade.

5.1.5 A compact “funding-adjusted alpha”

Many teams implement the decision rule by defining a single scalar:

$$\alpha_{t,H}^{\text{adj}} = \hat{r}_{t,H} - \text{sign}(\text{pos}) \cdot \hat{f}_{t,H} - \hat{c}_{t,H},$$

where $\text{sign}(\text{pos}) = +1$ for long and -1 for short. Then:

$$\text{Trade direction} = \text{sign}(\alpha_{t,H}^{\text{adj}}), \quad \text{Trade only if } |\alpha_{t,H}^{\text{adj}}| > \theta.$$

This is a useful mental model: funding is not a footnote; it is a direct subtraction from alpha for one side of the market and an addition for the other.

5.1.6 Signal families translated: momentum vs mean reversion

Directional signals differ mainly by *time scale* and *path*. Momentum expects continuation; mean reversion expects rebound. Funding awareness changes each in a distinct way.

5.1.7 Momentum: hold only when expected drift dominates funding drag

A basic momentum forecast over horizon H can be expressed as:

$$\hat{r}_{t,H} = \beta \cdot \frac{p_t - p_{t-L}}{p_{t-L}},$$

with lookback L and scaling β (often $\beta = 1$ if you treat the past return as a naive forecast).

5.1.8 Funding-aware momentum holding window

Momentum strategies often rely on *holding* through noise. With perps, the holding window is a control variable: if funding is strongly against you, you shorten H or you do not enter.

A practical rule:

$$H^* = \min \left(H_{\max}, \max \left(H_{\min}, \left\lfloor \frac{\hat{r}_{t,1}^+ - \hat{c}_{t,1}}{\max(\hat{f}_{t,1}, \epsilon)} \right\rfloor \right) \right),$$

where:

- $\hat{r}_{t,1}^+$ is your expected 1-hour return in the trade direction (clipped at zero),
- $\hat{f}_{t,1}$ is expected 1-hour funding cost in return units for that direction,
- ϵ avoids division by near-zero funding,
- $H_{\min}, H_{\max}$ are policy bounds.

Interpretation: when funding is expensive, the “breakeven time” compresses; you either harvest a fast move or you step aside.

5.1.9 Entry style

Momentum entries are vulnerable to buying tops/selling bottoms. Funding-aware entries reduce overtrading by requiring both:

1. a momentum trigger (e.g., return z-score, breakout),
2. a funding filter (e.g., “long only if funding is not too positive”).

This prevents a common failure mode: trend-following late into crowded longs where funding is highly positive, so even small pullbacks combined with funding drag erase the edge.

5.1.10 Mean reversion: funding can be your friend or your enemy

Mean reversion typically forecasts a price move *against* a recent displacement. A canonical signal uses a z-score:

$$z_t = \frac{p_t - \mu_t}{\sigma_t}, \quad \hat{r}_{t,H} = -\gamma z_t,$$

where μ_t and σ_t are a rolling mean and volatility estimate.

Funding awareness improves mean reversion in two ways:

1. **It selects which side to fade.** If funding is strongly positive (longs pay), then short mean-reversion trades get an additional tailwind from carry.
2. **It discourages “catching falling knives” when funding flips.** If funding turns strongly negative (shorts pay), then short mean-reversion trades require larger mispricings to justify paying carry while waiting for reversion.

5.1.11 A simple asymmetric threshold

Use different z-score triggers for long vs short:

$$\text{Go long if } z_t < -z_{\text{long}}, \quad \text{Go short if } z_t > z_{\text{short}},$$

and set

$$z_{\text{long}} = z_0 + k \cdot \max(f_t, 0), \quad z_{\text{short}} = z_0 + k \cdot \max(-f_t, 0),$$

where f_t is the current hourly funding rate. This makes the strategy *more conservative* on the side that must pay funding.

5.1.12 Position sizing under leverage: liquidation-aware, funding-aware

Once you have a trade direction, sizing is not “choose leverage.” It is: choose notional so that (i) the trade has a good risk-adjusted payoff *after* funding, and (ii) you can survive adverse moves without approaching maintenance constraints.

5.1.13 Sizing from a target risk budget

Let E_t be account equity allocated to the strategy, and let $\sigma_{t,H}$ be forecast volatility of return over horizon H . A standard risk budget sets position notional N so that a k -sigma move consumes at most a fraction η of equity:

$$N_t = \frac{\eta E_t}{k \sigma_{t,H}}.$$

Then convert notional to quantity: $q_t = N_t/p_t$ (up to contract conventions).

5.1.14 Funding-aware shrinkage

If the trade direction is expected to *pay* funding, reduce size in proportion to funding intensity:

$$N_t^{\text{adj}} = N_t \cdot \max\left(0, 1 - \lambda \frac{\max(\hat{f}_{t,H}, 0)}{\sigma_{t,H}}\right),$$

where λ controls how aggressively you penalize paying carry. Normalizing by σ is important: a funding rate that is huge in calm markets might be irrelevant in high volatility.

5.1.15 Hard caps: leverage & concentration limits

Even with volatility targeting, you still need hard constraints:

- **Max notional per asset:** to avoid one market dominating your liquidation risk.
- **Max gross notional:** to avoid correlated positions blowing up simultaneously.
- **Max leverage:** a policy cap lower than the venue maximum.

A useful sizing pipeline is:

1. Compute desired notional via the volatility-target notional.
2. Apply funding shrinkage.
3. Apply max-notional and max-leverage caps.
4. If caps bind frequently, reduce η or improve the forecast horizon match (rather than repeatedly trading clipped sizes).

5.1.16 Execution on Hyperliquid: making signal strength survive slippage

Funding-aware forecasting is pointless if the strategy gives back its edge in execution. The key is to tie *order type* to *urgency*.

5.1.17 Urgency ladder: ALO vs IOC vs TWAP

Hyperliquid supports time-in-force choices such as GTC, IOC, and ALO (post-only/add-liquidity-only), and also TWAP-style execution via the exchange endpoint. A practical mapping:

- **Low urgency (edge is small, costs matter):** enter with ALO limits to avoid paying spread and to earn maker rebates when available.

- **Medium urgency (edge moderate, risk rising):** use GTC limits near the microprice; accept some non-fills and re-quote.
- **High urgency (risk or signal decay):** use IOC to take liquidity, typically when inventory must be reduced or when a stop condition triggers.
- **Large size relative to depth:** use TWAP with a maximum slippage constraint; this replaces one large toxic print with a controlled schedule.

5.1.18 Funding-aware scheduling: align rebalancing to funding boundaries

Funding settles hourly. That creates a natural decision cadence:

- **Before the hour:** if your trade is paying funding and your alpha is marginal, flattening *before* the funding timestamp avoids an expected drag.
- **After the hour:** if your trade is receiving funding, you may allow a slightly wider exit band because carry compensates for some waiting (but do not let this become a justification for holding a broken thesis).

This is not market timing; it is cost control. The strategy still needs to earn its keep on price movement. Funding just changes the hurdle rate across the hour.

5.1.19 Concrete algorithms

This subsection provides two complete, implementable templates. They are intentionally conservative: they trade less when funding makes the edge fragile.

5.1.20 Algorithm 1: Funding-filtered momentum with time-decaying exposure

5.1.21 Inputs

- Prices: p_t (mid or microprice)
- Volatility estimate: $\sigma_{t,H}$
- Current hourly funding rate: f_t
- Lookback: L , holding horizon bounds: $H_{\min}, H_{\max}$
- Thresholds: momentum threshold τ , funding threshold $f_{\max}$, decision margin θ

5.1.22 Core logic

1. Compute momentum return $m_t = \frac{p_t - p_{t-L}}{p_{t-L}}$.
2. Direction proposal: $d_t = \text{sign}(m_t)$ if $|m_t| > \tau$, else flat.
3. Funding veto:
 - If $d_t = +1$ (long) and $f_t > f_{\max}$, set $d_t = 0$.
 - If $d_t = -1$ (short) and $f_t < -f_{\max}$, set $d_t = 0$.
4. Estimate $\alpha_{t,H}^{\text{adj}}$ (return forecast minus funding minus costs).
Trade only if $|\alpha_{t,H}^{\text{adj}}| > \theta$.
5. Set target horizon H^* shorter when funding is against you (as earlier).
6. Size notional via volatility targeting, then apply funding shrinkage.
7. Execute entry:
 - Use ALO if $|\alpha^{\text{adj}}|$ is just above threshold.
 - Use GTC or IOC if $|\alpha^{\text{adj}}|$ is large or volatility jumps.

8. During holding, decay exposure if funding turns sharply against the position:

$$q_{t+1} \leftarrow q_t \cdot \exp(-\kappa \max(0, \text{sign}(q_t) f_{t+1} - f_{\text{soft}})),$$

with a soft threshold f_{soft} and decay rate κ .

5.1.23 Why this works

Momentum is vulnerable to crowded positioning; crowdedness expresses itself in funding. The filter removes low-quality late entries, while the decay rule prevents a profitable trend trade from silently becoming a funding bleed when conditions flip.

5.1.24 Algorithm 2: Asymmetric mean reversion that “prefers” the funded side

5.1.25 Inputs

- Rolling mean and vol: μ_t, σ_t for z-score
- Funding rate: f_t
- Base entry threshold z_0 , funding sensitivity k
- Exit threshold z_{exit} (smaller than entry), stop threshold z_{stop} (larger than entry)

5.1.26 Core logic

1. Compute $z_t = (p_t - \mu_t)/\sigma_t$.
2. Compute side-specific thresholds:

$$z_{\text{long}} = z_0 + k \max(f_t, 0), \quad z_{\text{short}} = z_0 + k \max(-f_t, 0).$$

3. Entry:
 - If $z_t < -z_{\text{long}}$, open long.

- If $z_t > z_{\text{short}}$, open short.
- 4. Exit (take profit):
 - Close when $|z_t| < z_{\text{exit}}$ or when α^{adj} turns negative.
- 5. Stop:
 - If $|z_t| > z_{\text{stop}}$, reduce risk quickly (often IOC).
- 6. Sizing:
 - Volatility-target notional.
 - Apply *additional* shrinkage when you are on the side that pays funding.

5.1.27 Why this works

Mean reversion often wins by waiting for normalization; funding can be either a subsidy or a rent you pay while waiting. Asymmetric thresholds ensure you only pay rent when the mispricing is large.

5.1.28 Implementation checklist: making the strategy robust in production

Component	Robust practice on Hyperliquid perps
Funding inputs	Treat funding as an hourly state variable. Compute expected funding cost over your intended holding horizon and require alpha to clear that hurdle.
Costs model	Separate <i>spread/impact</i> from <i>fee schedule</i> . Choose ALO when you can, but do not let post-only cause chronic non-entry in fast markets.
Sizing	Use volatility-targeting as baseline; add funding-based shrinkage when paying. Enforce hard caps on notional and leverage.
Risk controls	Define: (i) maximum drawdown per day/session, (ii) maximum loss per trade, (iii) kill-switch when funding spikes beyond a policy bound against an open position.
Execution	Match order type to urgency: ALO (cheap), GTC (balanced), IOC (urgent), TWAP (large). Use reduce-only on exits and risk reductions.
State & reconciliation	Maintain a single source of truth for position, open orders, and remaining risk budget; reconcile against exchange state frequently, especially after bursts of cancels/modifies.

5.1.29 Pseudocode: a funding-aware directional execution loop

```
def directional_loop(state, market, params):
    # 1) Observe
    p = market.mid_price()
    f = market.current_hourly_funding_rate() # signed
    sigma = market.forecast_vol(params.H) # return vol over H

    # 2) Forecast (example: momentum)
    m = (p - market.mid_price_lag(params.L)) / market.mid_price_lag(params.L)

    # 3) Propose direction
    if abs(m) <= params.mom_threshold:
        desired_dir = 0
    else:
        desired_dir = 1 if m > 0 else -1

    # 4) Funding veto / asymmetry
    if desired_dir == 1 and f > params.funding_cap: # long pays too much
        desired_dir = 0
    if desired_dir == -1 and f < -params.funding_cap: # short pays too much
        desired_dir = 0

    # 5) Compute funding-adjusted alpha over horizon H (return units)
    r_hat = params.beta * m
    funding_hat = params.H * f
    cost_hat = market.estimated_roundtrip_cost_return_units()

    if desired_dir == 1:
        alpha_adj = r_hat - funding_hat - cost_hat
    elif desired_dir == -1:
        alpha_adj = -r_hat + funding_hat - cost_hat
    else:
        alpha_adj = 0.0

    if abs(alpha_adj) <= params.alpha_margin:
        desired_dir = 0

    # 6) Size with vol targeting + funding shrinkage
    N = (params.eta * state.equity) / (params.k * max(sigma, params.sigma_floor))

    paying_funding = (desired_dir == 1 and f > 0) or (desired_dir == -1 and f < 0)
    if paying_funding:
        N *= max(0.0, 1.0 - params.lambda_f * abs(f) / max(sigma, params.sigma_floor))

    N = min(N, params.max_notional)
    target_pos = desired_dir * (N / p)

    # 7) Execute: choose order type by urgency
    urgency = abs(alpha_adj) / max(sigma, params.sigma_floor)
    if desired_dir == 0:
        market.reduce_position_to_zero(state, method="IOC") # fast flatten if needed
    else:
        if urgency < params.urgency_low:
            market.work_limit_post_only_to_target(state, target_pos) # ALO
```

```

elif urgency < params.urgency_high:
    market.work_limit_to_target(state, target_pos) # GTC
else:
    market.cross_to_target(state, target_pos) # IOC

```

5.1.30 Common failure modes (and how funding-awareness prevents them)

1. **Overtrading small signals.** If the expected move is only a few basis points, funding plus spread often dominates. Funding-adjusted thresholds eliminate these trades by construction.
2. **Holding the wrong side through expensive funding.** Many discretionary traders notice funding only after PnL bleeds. The algorithms above treat funding as a first-class cost, so the strategy exits or shrinks before the drag accumulates.
3. **Leverage creep.** When a model has a modest edge, it is tempting to “make it work” with leverage. Volatility targeting and hard notional caps force the strategy to earn returns through forecasting and execution, not through fragility.
4. **Execution mismatch.** Entering with IOC when edge is small converts theoretical alpha into paid spread. The urgency ladder ties execution aggressiveness to funding-adjusted edge strength.

5.2 Funding and Carry Optimization Strategies

Directional strategies (previous section) treat funding as a hurdle that can suppress marginal trades. Carry strategies flip the perspective: funding becomes the *primary* return stream, and price movement becomes a risk factor to neutralize. The goal is not to “be right on direction,” but to structure a position whose PnL is dominated by predictable carry while tightly controlling the ways the hedge can break.

On Hyperliquid, two venue-specific facts drive the design:

- Funding is **paid hourly, peer-to-peer**, and the protocol does **not take a fee** from funding transfers.
- Funding notional conversion uses the **oracle price** (not the mark), so your expected hourly cashflow is tied to oracle-referenced notional exposure.

This section focuses on the canonical carry trade (long spot, short perp when funding is positive), and then generalizes to robust execution, basis drift controls, margin/liquidation guardrails, and action-budget-aware automation.

5.2.1 Delta-neutral framing: turn funding into a targeted yield stream

The simplest funding-harvest structure is a *delta-neutral* pair:

- **Long spot** +1 unit of the asset (held on a spot venue, wallet, or elsewhere).
- **Short perp** -1 unit of the same asset on Hyperliquid.

Let S_t be the spot price and P_t be the perp price (both in USD terms for intuition). Denote the perp position quantity as q (negative for a short). A first-order PnL decomposition over a short horizon is:

$$\Delta\Pi \approx (+q_s) \Delta S + (q_p) \Delta P + \Pi_{fund} - \Pi_{cost}.$$

If you set $q_s = -q_p$ and if $\Delta S \approx \Delta P$ (prices move together), then the price terms largely cancel and the residual return comes from funding net of costs.

Expected hourly funding PnL (sign check). If the hourly funding rate is f_t (positive means longs pay shorts), then for a *short* perp position ($q_p < 0$) your expected hourly funding cashflow is positive:

$$\mathbb{E}[\Pi_{fund,t \rightarrow t+1}] \approx -q_p \cdot f_t \cdot P_t^{oracle}$$

(oracle-referenced notional). This oracle reference matters operationally: you should model funding cashflows using oracle notional even if you monitor mark/mid for execution.

5.2.2 Basis drift risk: the hedge breaks through $P_t - S_t$

Delta neutrality is not binary; it is a *tracking problem*. The dominant risk is that spot and perp prices diverge (basis moves) or that the hedge ratio becomes imperfect due to execution, fees, or delays.

Define the basis:

$$B_t = P_t - S_t.$$

If you hold +1 spot and -1 perp, your exposure to the basis is approximately:

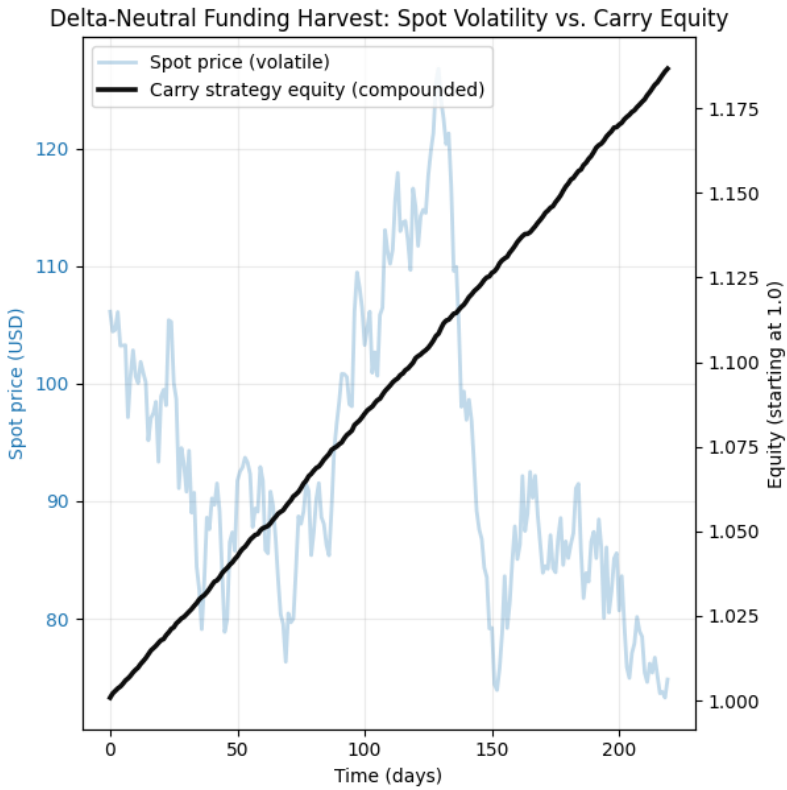
$$\Delta \Pi_{basis} \approx -\Delta B_t.$$

So if the perp price rises relative to spot (basis widens upward), the short-perp leg loses relative to the spot leg.

What makes basis drift nontrivial on perps. On Hyperliquid, funding is computed from a premium index that uses *impact bid/ask prices relative to oracle*, sampled frequently and averaged, with an interest-rate component that is *clamped*. Two implications follow:

- **Funding and basis are linked but not identical.** A high funding rate often corresponds to a positive premium, but the relationship is mediated by impact prices and clamps.
- **The clamp changes tail behavior.** When the interest-rate component is capped, extreme conditions can show up as basis volatility rather than purely as an extreme funding print, so you still need explicit basis risk controls.

Because basis drift can swamp a few hours (or even days) of funding income, a carry strategy is best viewed as *sell a stream of funding cash-flows while running a tight basis-risk book*.



5.2.3 Designing the carry book: what you optimize

A funding strategy is not “short the perp when funding is positive.” The implementable objective is:

$$\max \mathbb{E}[\text{Funding}] - \mathbb{E}[\text{Hedge Costs}] - \text{BasisRiskPenalty} - \text{LiquidationRiskPenalty}$$

Each term is actionable:

- **Funding:** depends on your notional and the hourly rate; model it on oracle notional.

- **Hedge costs:** spot and perp entry/exit costs, plus any ongoing re-hedge costs.
- **Basis risk:** controlled by basis limits, re-hedge logic, and asset selection.
- **Liquidation risk:** controlled by leverage, maintenance margin buffer, and drawdown rules on the perp leg.

5.2.4 A concrete baseline strategy: long spot / short perp carry

Assume you can hold spot and trade perps. When funding is *persistently positive*, a baseline carry book is:

1. Buy spot quantity Q (or allocate an existing spot holding as the hedge asset).
2. Short perp quantity Q on Hyperliquid.
3. Maintain delta neutrality by periodically rebalancing Q when inventory drifts.

5.2.5 Sizing: notional first, then quantity

Choose a target perp notional N based on (i) the stability of funding, (ii) your risk tolerance for basis, and (iii) liquidation buffers.

A practical sizing heuristic:

$$N = \min \left(N_{max}, \frac{E \cdot \eta}{\sigma_B} \right),$$

where:

- E is equity allocated to this book (USDC terms),
- η is a basis-risk budget (fraction of equity you can tolerate losing to basis moves),

- σ_B is an estimate of basis volatility (or a conservative proxy derived from perp–spot spread variance).

Then $Q = N/S_t$ spot units and short Q units on perps. If your hedge venue uses different contract sizing, convert carefully; the goal is matching *delta*, not just notional.

5.2.6 Entry filter: fund-and-premium consistency

Funding can spike briefly, but the trade only works if you can hold it through multiple funding hours without being shaken out by basis widening.

A conservative entry filter requires:

$$\hat{f}_{t:t+K} > f_{min} \quad \text{and} \quad |B_t| < B_{max},$$

where $\hat{f}_{t:t+K}$ is the expected average funding over the next K hours. Interpretation:

- If the perp is already extremely expensive versus spot (B_t large), you may be late; a basis snapback can erase funding.
- If funding is positive but noisy, you need enough expected duration (K hours) to amortize entry costs.

5.2.7 Re-hedging and action budgets: staying neutral without overtrading

Carry books are often ruined by *over-hedging*: you churn fees and cross spreads to maintain an unrealistic idea of perfect neutrality. On Hyperliquid, this is compounded by the fact that **address-based action limits** can implicitly cap how often you can update orders or re-balance.

A robust approach is *banded re-hedging*:

- Define a delta band: $\Delta \in [-\Delta_{max}, +\Delta_{max}]$.

- Only rebalance when your net delta exits the band.
- When rebalancing, trade only enough to return to the *center* (or a tighter inner band), not to zero every time.

Why bands work. They convert continuous micro-fluctuations into infrequent, meaningful actions. This is both cost-efficient and compatible with action limits.

5.2.8 A simple delta estimator

Let Q_s be spot units and Q_p be perp units (positive long, negative short). Approximate net delta (in units) as:

$$\Delta_{units} = Q_s + Q_p.$$

You can scale to USD delta by multiplying by an appropriate price (spot, mid, or oracle). The key is consistency: use the same convention for the band and the trigger.

5.2.9 Basis risk controls: limits, stops, and “basis VaR”

Because basis drift is the primary risk, carry books need explicit basis controls, not just generic PnL stops.

5.2.10 Hard basis limit

Set a maximum tolerable basis move against your position:

$$|B_t - B_{entry}| > B_{stop} \Rightarrow \text{de-risk.}$$

De-risk can mean:

- reduce perp size (lower liquidation pressure),
- close both legs (realize basis loss and stop the bleed),
- temporarily over-hedge (rare; only if you have strong reason and low costs).

5.2.11 Basis VaR approximation

A quick risk proxy is:

$$\text{VaR}_B(\alpha) \approx z_\alpha \cdot \sigma_B \cdot N,$$

where z_α is a normal quantile and N is your notional. This is a coarse model, but it forces you to scale down when basis volatility rises.

5.2.12 Margin and liquidation constraints on the perp leg

The carry book is delta-neutral, but it is not *margin-neutral*. The perp leg can be liquidated if the perp price moves against you enough to breach maintenance conditions. Hyperliquid liquidations begin with market orders to close the position via the order book; if equity drops below a deeper threshold without successful closure, backstop liquidation occurs via a liquidator vault component strategy of HLP.

For a short-perp carry book, the risk event is a fast upward move in the perp (often alongside spot), plus any basis widening that makes the perp outperform spot. Even if spot rises (helping your spot leg), liquidation can still occur if the perp leg lacks sufficient margin to survive the path.

5.2.13 A practical leverage policy for carry

Carry returns are usually small per hour; they do not justify thin buffers. A defensible policy is:

- Use leverage well below the maximum allowed.
- Maintain a buffer so that a multi-sigma basis widening does not force liquidation before you can react.
- When funding is favorable but basis volatility rises, *reduce notional* rather than increasing leverage.

5.2.14 Margin-aware re-hedge priority

When risk rises (basis widening, volatility spike), prioritize actions that *reduce liquidation probability* even if they reduce funding income:

1. Reduce perp short notional (lower margin stress).
2. Then adjust spot to restore delta neutrality if desired.

This ordering is conservative but matches the asymmetry of failure: liquidation is a discontinuity.

5.2.15 Execution details: minimizing hedge slippage

Carry strategies are fundamentally execution-sensitive because they trade at least two legs.

5.2.16 Sequencing and “leg risk”

If you cannot execute both legs simultaneously, you are exposed to price changes between legs. Two practical tactics:

- **Risk-first sequencing:** put on the leg that reduces your worst-case exposure first. For example, if you already hold spot and are adding the short perp hedge, enter the perp promptly; if you are opening from flat, avoid being naked short perp without spot unless you intend that risk.
- **Staggered sizing:** enter in tranches (e.g., 25% increments), rechecking basis and liquidity each tranche.

5.2.17 Order types as cost controls

Tie order aggressiveness to basis conditions:

- If basis is stable and liquidity is deep, work maker-style orders (e.g., post-only/add-liquidity-only where appropriate) to reduce costs.
- If basis is moving quickly against you, switch to more aggressive execution to reduce leg risk.
- For large size, prefer scheduled execution (TWAP with slippage constraints) to avoid paying extreme impact all at once.

These are strategy-level rules; the exact API payloads and signing are handled via the POST /exchange lifecycle described earlier.

5.2.18 Carry selection across assets: harvesting asymmetry without concentration

In practice, the best carry opportunities are not constant. Funding distributions differ by asset and by regime. A robust allocator treats carry as a cross-sectional portfolio:

1. Estimate **expected net funding** per asset (after conservative costs).
2. Penalize assets with **high basis volatility** or fragile liquidity.
3. Cap concentration so one asset's basis event cannot dominate the book.

A simple scoring function for asset i :

$$\text{Score}_i = \hat{f}_i - a \hat{c}_i - b \sigma_{B,i},$$

with weights $a, b > 0$. Allocate notional proportional to $\max(\text{Score}_i, 0)$, subject to per-asset caps.

5.2.19 Operational blueprint: a carry bot that respects risk and action limits

Below is a reference loop that is realistic for production: it uses bands (fewer actions), explicit basis stops, and hourly funding cadence.

```

def carry_loop(state, market, hedge, params):
    """
    market: Hyperliquid perp venue data/execution wrapper
    hedge: spot venue or wallet/position tracker
    params: risk bands, basis limits, action budget
    """

    # Observe
    S = hedge.spot_price()
    P = market.perp_mid_price()
    f = market.current_hourly_funding_rate() # signed hourly rate
    B = P - S

    # Positions (units)
    Qs = hedge.spot_units() # + long spot
    Qp = market.perp_units() # - short perp

    # Entry condition (only if flat-ish)
    if abs(Qs + Qp) < params.delta_enter_tol and abs(Qp) < params.perp_enter_tol:
    if f > params.funding_min and abs(B) < params.basis_max:
    target_notional = min(params.max_notional, params.equity * params.notional_frac)
    Q_target = target_notional / S

    # Enter in tranches to reduce leg risk and impact
    for k in range:
    q_k = (Q_target / params.tranches)

    hedge.buy_spot(q_k) # or allocate existing spot if already held
    market.sell_perp(q_k) # open short perp hedge

    if abs((market.perp_mid_price() - hedge.spot_price()) - B) > params.
        basis_jump_abort:
    break

    # Risk controls: basis stop
    if state.basis_entry is not None:
    if abs(B - state.basis_entry) > params.basis_stop:
    # De-risk: close perp first to reduce liquidation chance, then spot
    market.reduce_perp_toward(0.0, urgency="high")
    hedge.reduce_spot_toward
    state.reset()

    # Re-hedge using a delta band to save fees and actions
    delta_units = Qs + Qp
    if abs(delta_units) > params.delta_band:
    # Trade back toward the center (not necessarily to zero)
    adjust = -np.sign(delta_units) * min(abs(delta_units), params.delta_recenter_size)

    # Prefer adjusting perp (faster, often cheaper to trade than moving spot)
    market.adjust_perp_units(adjust, urgency="medium")

    # Hour-boundary logic: if paying funding, tighten; if receiving, allow bands
    if market.is_close_to_funding_timestamp():
    if (Qp < 0 and f < 0) or (Qp > 0 and f > 0): # paying on the perp side
    market.reduce_perp_toward(Qp * params.pre_funding_shrink, urgency="high")

```


The key architectural idea is that carry is *not* high-frequency scalping. The loop avoids constant micro-adjustments, which both reduces fees and keeps you within address-based action limits.

5.2.20 Checklist: what to log and monitor

Carry strategies fail silently unless you log the right state variables. The following is the minimum viable monitoring set.

Metric	Why it matters
Hourly funding rate f_t (per asset)	Core revenue driver; regime shifts quickly when positioning flips.
Oracle price and funding notional	Funding cashflows are oracle-notional-based; track what you are actually paying/receiving against.
Basis $B_t = P_t - S_t$ and ΔB_t	Primary risk factor; basis shocks can dominate many hours of funding.
Net delta band breaches	Indicates hedge drift; also a proxy for how often you'll need to spend action budget.
Perp margin buffer / liquidation distance	Carry returns do not compensate for thin buffers; margin stress should trigger de-risking.
Net PnL attribution (funding vs basis vs fees)	Confirms the strategy is earning via carry rather than taking disguised direction.

5.2.21 Takeaways

Funding and carry optimization on Hyperliquid is a structured exercise in earning hourly peer-to-peer funding while engineering away the failure modes:

- Delta-neutrality is a *process*, not a static setting; basis drift is the central risk.
- Model funding cashflows on *oracle-referenced notional*, and treat clamps/impact-price construction as reasons to keep basis controls explicit.
- Use banded re-hedging to reduce fees and fit within address-based action limits.
- Keep leverage conservative: liquidation is a discontinuity that can erase months of carry.

The next section extends these ideas to *basis and arbitrage execution* across venues, where sequencing risk and latency become first-order constraints rather than implementation details.

5.3 Basis and Arbitrage Execution

The previous section treated basis risk as the main hazard in funding carry. Arbitrage strategies take the next step: rather than merely *controlling* basis drift while collecting funding, you attempt to *moneitize* temporary mispricings between (i) Hyperliquid perps and external spot/perp venues, or (ii) Hyperliquid perps and an internal reference such as oracle-based notions of “fair” pricing.

The intellectual trap is to treat arbitrage as “risk-free.” In practice, nearly all of the risk comes from *execution sequencing*: you rarely fill both legs at the same time and the window between leg executions is exactly when the mispricing can vanish or reverse. On Hyperliquid, this is shaped by:

- **Premium definition:** mispricing is not only last trade vs oracle; Hyperliquid’s premium concept uses *impact prices relative to oracle*. A small last-trade premium can still imply poor arbitrage if impact prices are wide.
- **Order book constraints:** tick/lot constraints and time-in-force choices (IOC vs post-only/add-liquidity-only) determine whether you can actually capture the spread.
- **Rate-limit engineering:** message/weight limits and address-based action limits force you to budget cancels/modifies and limit how aggressively you can chase the book.
- **Different clocks:** funding settles hourly, while premium inputs are sampled at higher frequency; the opportunity set can change intrahour even if you plan around the hourly boundary.

This section builds a practical arbitrage framework around a single question:

Can I execute both legs with enough certainty and speed that the expected convergence profit survives fees, slippage, and the risk of being temporarily unhedged?

5.3.1 The object you trade: basis as a priced risk

Let P_t^{HL} be the Hyperliquid perp price (mid for intuition), and S_t^{EXT} be an external spot price (or an external perp price adjusted to the same funding/carry convention). Define the cross-venue basis:

$$B_t = P_t^{HL} - S_t^{EXT}.$$

A classic convergence trade is:

- If B_t is **too high**: short Hyperliquid perp, long external spot.
- If B_t is **too low**: long Hyperliquid perp, short external spot (or short external perp).

5.3.2 From “mid basis” to basis-at-fill

Arbitrage is executed, not observed. Replace mid prices with *expected fill prices*:

$$\tilde{B}_t = \tilde{P}_t^{HL} - \tilde{S}_t^{EXT},$$

where each tilde price includes:

- the spread you will cross (if using aggressive orders),
- expected slippage/impact for your size,
- fees (taker fees or maker rebates) translated into price units.

Because Hyperliquid’s premium index depends on *impact prices vs oracle*, a good rule is: *if your expected fill is close to the impact price used in premium calculations, your model of convergence will be much more reliable*. If your fills are far worse than impact assumptions, the apparent edge is often illusory.

5.3.3 Profit model: convergence plus carry, minus frictions

For a two-leg arbitrage with quantities matched in units Q (positive Q is long spot / short perp in the $B_t > 0$ case), the expected PnL over a short horizon H can be expressed as:

$$\mathbb{E}[\Pi] \approx Q \cdot \mathbb{E}[\Delta B] + \mathbb{E}[\Pi_{fund}^{HL}] - \mathbb{E}[\Pi_{fees+slip}].$$

Key points:

- **Convergence term** $Q \cdot \Delta B$: this is the pure arbitrage component.
- **Funding term**: you may pay or receive funding while holding the Hyperliquid perp leg; since funding settles hourly, this term is naturally discretized even if you rebalance intrahour.
- **Friction term**: this includes both venues and is usually the largest model risk. Underestimating friction turns “arb” into a slow bleed.

A conservative execution system treats the arbitrage expected-value condition as an entry gating condition:

$$\mathbb{E}[\Pi] > 0 \Rightarrow \text{attempt}; \quad \mathbb{E}[\Pi] \leq 0 \Rightarrow \text{skip}.$$

5.3.4 Execution sequencing risk: the arb lives in the time gap

The central engineering problem is the *time delta* between the first fill and the second fill. During that gap you are exposed to price changes, basis changes, and, in stressed conditions, liquidity evaporation.

5.3.5 Leg risk taxonomy

There are three common execution patterns, each with distinct risk:

1. **Aggressive–Aggressive:** cross the spread on both venues for fast completion. Lowest sequencing risk, highest cost.
2. **Passive–Aggressive (maker then taker):** place a maker order on the expensive venue, hedge aggressively when it fills. Lower cost, but you may never get filled, or you may get filled at exactly the wrong time.
3. **Passive–Passive:** try to earn rebates on both legs. Rarely viable for arbitrage unless the mispricing is persistent and deep, because completion probability becomes the limiting factor.

For most real systems, the best default is *Passive on Hyperliquid when you have time, Aggressive when you do not*. Hyperliquid's post-only/add-liquidity-only option is valuable for cost control, but it should be treated as a *conditional* choice, not a religion.

5.3.6 Cross-venue execution risk

An actionable way to think about sequencing is as a state machine: detect edge, place first leg, wait for confirmation, then place hedge leg, and continuously monitor whether the expected profit still exists. The important part is that the *risk window* begins at the first fill, not at order submission.

5.3.7 Two guardrails for sequencing

5.3.8 Guardrail 1: timeouts and edge decay

If the first leg does not fill quickly, cancel and recompute. The correct timeout is not arbitrary; it is related to how quickly your edge estimate decorrelates. A simple rule is:

$$\text{timeout} = \min \left(T_{max}, \frac{\kappa}{\hat{\sigma}_B} \right),$$

where $\hat{\sigma}_B$ is an intraday estimate of basis volatility and κ sets how much basis movement you are willing to tolerate while waiting.

5.3.9 Guardrail 2: dynamic hedge urgency

Once the first leg fills, the hedge leg becomes a risk-reduction trade. Use the most reliable execution mode available to minimize the unhedged interval, even if it is more expensive. Paying a bit more is often cheaper than being exposed during a basis snap.

5.3.10 Practical arbitrage patterns

This section focuses on patterns that translate into robust systems rather than fragile theoretical trades.

5.3.11 Pattern A: Cross-venue spot–perp convergence

Setup.

- External venue: spot position in the asset.
- Hyperliquid: perp leg.

Trade. When Hyperliquid perp is rich to external spot (after expected execution costs), do:

1. **Sell** Hyperliquid perp (often maker/post-only if time allows; IOC if edge is fleeting).
2. **Buy** external spot aggressively enough to complete.

Close. Exit when basis returns toward a target band, or when the net expected PnL from holding further (including funding on the perp leg) becomes negative.

Key risk. External spot settlement/transfer constraints can make “spot” not truly fungible at the timescale you need. Even if prices converge, operational frictions can trap capital. Treat this as part of $\Pi_{\text{fees}+\text{slip}}$.

5.3.12 Pattern B: Cross-venue perp–perp spread

If both venues list perps, you can trade a spread that is more symmetric operationally:

- Long the cheap perp, short the rich perp.

This reduces the need to custody spot, but introduces a new complexity: you now have two funding streams and two premium mechanisms. Because Hyperliquid’s funding/premium uses impact prices vs oracle, while another venue may use a different mark mechanism, the spread can be driven by funding mechanics rather than pure mispricing. Your model must include both carry terms.

5.3.13 Pattern C: Intravenue proxy basis using oracle and impact

Even without an external venue, you can treat the Hyperliquid perp vs oracle as a proxy mispricing:

$$\text{PremiumProxy}_t \approx \frac{P_t^{\text{impact}} - P_t^{\text{oracle}}}{P_t^{\text{oracle}}}.$$

The edge is not “arb” in the strict cross-venue sense, but a systematic bet that premium mean-reverts because the funding rule acts as a feedback mechanism. The same sequencing ideas apply (you still must execute into the book), but the hedge may be a different asset or a reduced-risk sizing rather than a literal spot hedge.

5.3.14 Choosing order types under microstructure and rate limits

The biggest practical mistake is to run an arbitrage strategy with the order management style of a market maker: endless quote updates, rapid cancel/replace loops, and micro-optimizations. Arbitrage needs *completion*, not perfect price.

5.3.15 IOC vs post-only as a completion dial

A workable policy is:

- Use post-only/add-liquidity-only when:
 1. edge is large enough to wait,
 2. you have a hedge already in place (so you are not naked),
 3. order book depth is stable.
- Use IOC when:
 1. the first leg has filled and you must hedge,
 2. liquidity is thinning or volatility is spiking,
 3. you are approaching an hourly funding boundary where the carry term changes the trade economics.

5.3.16 Action-budget-aware batching

Hyperliquid API batching reduces IP-weighted request usage, but **does not reduce address-based action consumption**. That means batching is good for network efficiency, but it is not a license to place more cancels/modifies. For arbitrage systems, this has two direct design consequences:

1. **Fewer, higher-conviction attempts:** increase your entry threshold so you do not spray orders for marginal edges.
2. **Stateful order management:** prefer “place once, then either fill or cancel on timeout” rather than repeatedly inching the price.

5.3.17 Rebalancing around hourly funding: discrete constraints, continuous signals

Funding settles hourly, but premium sampling is higher-frequency. This creates a useful timing heuristic:

- **Intrahour:** prioritize convergence and hedge completion; do not overfit to small funding changes.
- **Near the hour:** re-evaluate whether holding through the next funding event improves or worsens expected value. If you expect to *pay* funding on your Hyperliquid leg and convergence is largely realized, flattening just before the timestamp can improve realized performance.

The important nuance is that hourly funding is a *stepwise cashflow*, while basis can move continuously. Your strategy should not force trades exactly on the hour; it should simply recognize that the carry term changes discontinuously there.

5.3.18 A robust arbitrage decision engine

A production-quality decision engine typically has three layers:

5.3.19 Layer 1: Edge estimation in fill space

Compute edge using expected fill prices and include fees:

$$\hat{\Pi}_{edge} = Q \cdot \left(\tilde{P}_{sell}^{HL} - \tilde{S}_{buy}^{EXT} \right).$$

5.3.20 Layer 2: Sequencing penalty

Add an explicit penalty for being exposed during Δt :

$$\text{Penalty}_{\Delta t} = \lambda Q \hat{\sigma}_B \sqrt{\Delta t},$$

with λ calibrated so that you only trade when the mispricing is large relative to realistic basis movement during the hedge window.

5.3.21 Layer 3: Funding term (if holding)

If the trade is expected to remain open across at least one hourly boundary, include expected funding on the Hyperliquid perp leg:

$$\hat{\Pi}_{fund} \approx -q \sum_u \hat{f}_u P_u^{oracle}.$$

Then gate:

$$\hat{\Pi}_{edge} + \hat{\Pi}_{fund} - \text{Penalty}_{\Delta t} > 0.$$

This structure makes a crucial point tangible: arbitrage is not purely about price discrepancy; it is about discrepancy *relative to* execution risk and carry.

5.3.22 Operational checklist: what breaks arbs in practice

Issue	Mitigation
Edge measured at mid, executed at impact	Compute edge with expected fill prices; use impact-price-aware slippage models.
Hedge leg fails or is delayed	Use timeouts; switch hedge leg to aggressive execution after first fill.
Too many cancel/replace actions	Raise entry threshold; use “attempt–timeout–cancel” loops; remember batching does not reduce address actions.
Hourly funding flips trade economics	Include expected funding if holding across hour; reassess near funding timestamps.
Basis moves intrahour despite stable funding	Treat basis volatility as primary; model σ_B and impose a sequencing penalty.
Tick/lot constraints prevent precise hedges	Hedge within a band; size tranches to match discrete increments and avoid chronic residual exposure.

5.3.23 Takeaways

Basis and arbitrage execution on Hyperliquid is a systems problem dressed up as a pricing problem:

- Model the opportunity in *fill space* (impact-aware), not at the mid.

- Treat the time gap between legs as the core risk; charge a sequencing penalty and enforce timeouts.
- Use IOC when hedging is urgent, and use post-only selectively when you can afford to wait.
- Engineer around rate limits: batching helps IP-weighted usage, but address-based action budgets still bind, so reduce churn.
- Remember that funding is hourly but premium can change faster; plan around the boundary without becoming hostage to it.

With these execution-first principles in place, you can approach arbitrage as a repeatable, risk-budgeted strategy rather than a collection of brittle one-off trades.

5.4 Market Making Edge Decomposition

Directional and carry systems start from an explicit view of “fair value” and then pay execution costs to express that view. Market making flips the posture: you *sell immediacy* to others by quoting both sides, hoping that (i) you capture spread and rebates often enough, and (ii) the losses from trading against better-informed or faster participants do not overwhelm those gains.

Because that trade-off is subtle, professional market makers do not evaluate a quoting strategy by raw PnL alone. They decompose returns into components that map directly to controllable levers: how wide you quote, how often you are maker vs taker, how you manage inventory, and how “toxic” the incoming flow is. This section builds a formal expected-value (EV) model for continuous liquidity provision on Hyperliquid perps that explicitly includes:

- **Realized spread capture** (what you earn when you buy then later sell, or vice versa),
- **Maker rebates / fees** (a systematic edge term on Hyperliquid, often nontrivial),

- **Toxicity losses** (losses from price moves after you get filled, typically measured by markout),
- **Inventory risk and funding coupling** (residual inventory creates variance and can add funding drag or benefit).

5.4.1 The core unit: a single maker fill

Start at the atomic event: your resting limit order is hit and you trade as maker.

Let the midprice at the moment of the fill be m_t . Assume you post a bid at $b_t = m_t - \delta_b$ and/or an ask at $a_t = m_t + \delta_a$. Define:

- Half-spreads: $\delta_b \geq 0, \delta_a \geq 0$
- Fill indicator variables: $I_t^b, I_t^a \in \{0, 1\}$ (did you get filled on bid or ask?)
- Trade size (in base units): q (assume constant for now)

A bid fill increases inventory by $+q$, an ask fill decreases inventory by $-q$.

5.4.2 Immediate PnL vs subsequent PnL

At the instant of a maker fill, the trade is executed *inside* the spread (relative to mid), so there is an immediate “paper gain” relative to mid:

$$\Pi_t^{imm} = q (I_t^a \delta_a + I_t^b \delta_b).$$

However, this is not what you actually keep. The midprice can move after the fill. If the mid moves against your new inventory, you lose. That loss is the essence of toxic flow.

To formalize it, define a *markout horizon* τ (e.g., a few seconds to a few minutes), and let the mid at $t + \tau$ be $m_{t+\tau}$. The signed inventory change from the trade is:

$$\Delta Q_t = q(I_t^b - I_t^a).$$

The markout PnL over horizon τ attributable to price movement is:

$$\Pi_t^{mo}(\tau) = \Delta Q_t \cdot (m_{t+\tau} - m_t).$$

If incoming flow is informed (or simply faster), then after you buy, price tends to fall, and after you sell, price tends to rise. In that case $\mathbb{E}[\Pi_t^{mo}(\tau)] < 0$.

5.4.3 From micro to macro: a PnL decomposition over time

Aggregate over many fills in a window $[0, T]$. A practical decomposition is:

$$\Pi_{0:T} = \Pi_{spread} + \Pi_{fees} - \Pi_{tox} + \Pi_{inv} + \Pi_{fund}.$$

Not every desk uses the same labels, but this structure is stable: two positive engines (spread, rebates), and several negative terms (toxicity, inventory variance costs, sometimes funding drag).

5.4.4 Realized spread: what you actually capture

The theoretical half-spread δ is not the realized half-spread. The realized spread is defined over round trips: buy then sell (or sell then buy). A simple estimator is:

$$\Pi_{spread} \approx \sum_{k \in \mathcal{RT}} q_k \left(p_k^{sell} - p_k^{buy} \right),$$

where $\mathcal{RT}$ indexes matched buy/sell pairs that unwind inventory. If you do not unwind (inventory accumulates), the spread term becomes hard to separate from inventory PnL, which is why inventory control is inseparable from edge measurement.

5.4.5 Fees and maker rebates: a systematic term on Hyperliquid

On Hyperliquid, maker rebates can be a material contributor. Conceptually, you can represent fee PnL as:

$$\Pi_{\text{fees}} = \sum_{j \in \text{trades}} (-\text{feeRate}_j) \cdot \text{notional}_j,$$

with feeRate negative for rebates and positive for fees (sign convention). Because fees are assessed by tiering and maker rebates are paid continuously per trade, this term behaves like an edge multiplier: if you can safely increase maker volume without inviting toxic fills, rebates can raise EV even when spreads are tight.

The practical lesson is that quoting decisions should be evaluated in net-of-fees spread units:

$$\delta^{\text{net}} = \delta + \text{rebatePerNotional} \times m_t,$$

so that a narrow quote can still be profitable if rebates are high enough and toxicity is low.

5.4.6 Toxicity: measuring the hidden loss with markout

The most implementable toxicity metric is the average markout per unit notional:

$$\text{Markout}(\tau) = \frac{\sum_t \Pi_t^{\text{mo}}(\tau)}{\sum_t |q_t| m_t}.$$

Interpretation:

- **Near zero or positive:** your fills are not systematically followed by adverse moves; flow is relatively benign.
- **Negative:** you are consistently trading just before the price moves against you; your quotes are being “picked off” or you are leaning too hard in fast markets.

A common mistake is to compute toxicity only at a single horizon τ . In practice, use a small grid (e.g., $\tau \in \{1\text{s}, 5\text{s}, 30\text{s}, 2\text{m}\}$) to distinguish:

- **Latency-type pickoffs** (very short-horizon negative markout),
- **Information-type trades** (negative markout persists across minutes),
- **Inventory drift** (markout depends strongly on your own skewing and position).

5.4.7 A compact EV model for continuous quoting

To connect decomposition to design, you want an EV expression that depends on quote width, fill intensity, and toxicity. A standard simplification models buy and sell market order arrivals as Poisson processes with intensities $\lambda_b(\delta_b)$ and $\lambda_a(\delta_a)$ that fall as you quote wider.

Assume symmetric quoting for clarity: $\delta_b = \delta_a = \delta$, and equal intensities $\lambda(\delta)$. Over a small time step Δt , the expected spread capture is:

$$\mathbb{E}[\Delta\Pi_{spread}] \approx 2\lambda(\delta) \Delta t \cdot q\delta.$$

Expected fee/rebate contribution scales with notional traded:

$$\mathbb{E}[\Delta\Pi_{fees}] \approx 2\lambda(\delta) \Delta t \cdot (rebatePerNotional) \cdot qm_t.$$

Expected toxicity cost can be represented by a negative expected markout per fill:

$$\mathbb{E}[\Delta\Pi_{tox}] \approx 2\lambda(\delta) \Delta t \cdot q \cdot \mathbb{E}[m_{t+\tau} - m_t \mid fill]_{signed}.$$

The conditional expectation in the toxicity-adjusted expected value is exactly what markout estimates. If markout at horizon τ is $-\mu_{mo}$ (a positive number $\mu_{mo} > 0$ representing loss per unit), then:

$$\mathbb{E}[\Delta\Pi_{tox}] \approx 2\lambda(\delta) \Delta t \cdot q\mu_{mo}.$$

Combine the three:

$$\mathbb{E}[\Delta\Pi] \approx 2\lambda(\delta) \Delta t \cdot q (\delta + rebatePerNotional \cdot m_t - \mu_{mo}) - InventoryRiskCost.$$

The market-maker expected-value expression is the key design inequality for market making:

$$\delta + rebatePerNotional \cdot m_t > \mu_{mo}.$$

This inequality holds before inventory risk is considered. If your mark-out loss per fill exceeds your net spread plus rebates, you do not have a market making edge at that configuration. You can regain EV by:

- quoting wider (increase δ),
- improving queue position/latency to reduce μ_{mo} ,
- reducing size q in toxic periods,
- gating quotes using volatility or order-flow signals,
- leaning inventory intelligently so you are not repeatedly filled in the wrong direction.

5.4.8 Adding funding: inventory has a carry term on perps

On perps, inventory is not just a price-risk exposure; it also has a funding stream. Recent work (and practical experience) shows this can meaningfully couple to market making PnL.

Let Q_t be your inventory (positive long, negative short). Funding is paid hourly and is based on oracle-referenced notional; represent the hourly funding rate as f_t (positive means longs pay). Over one hour:

$$\Pi_{fund,t \rightarrow t+1} \approx -Q_t \cdot f_t \cdot P_t^{oracle}.$$

Two implications matter for strategy design:

- **Funding changes the “fair” inventory.** If funding is strongly positive, being short inventory is not only a hedge; it may also earn carry. Conversely, being long inventory has an added drag.
- **Funding can masquerade as market making skill.** A bot that tends to run short during positive funding can show steady profits even if its spread capture is mediocre. Without decomposition, you may mistakenly attribute this to quoting quality.

So a complete attribution report should separate Π_{fund} from Π_{spread} and Π_{tox} , even if all three are realized through the same stream of fills.

5.4.9 Estimating components in practice (without fragile assumptions)

A decomposition is only useful if you can estimate it reliably. The following workflow is practical for on-chain perps:

5.4.10 Step 1: Label trades as maker vs taker and store microstate

For each fill, store:

- side (buy/sell), size, price,
- maker/taker flag,
- midprice at fill, plus midprice at several markout horizons,
- current funding rate (for later attribution),
- inventory before/after.

Even if your final model is sophisticated, this minimal dataset prevents you from guessing later.

5.4.11 Step 2: Compute markouts (toxicity) on a horizon grid

For each fill t , compute:

$$\Pi_t^{mo}(\tau) = \Delta Q_t(m_{t+\tau} - m_t),$$

then aggregate by regime (volatility buckets, funding buckets, time-of-day). This produces a toxicity surface that tells you when your quotes are being punished.

5.4.12 Step 3: Compute net spread in dollars

For maker trades, estimate:

$$\Pi_{netSpread} \approx \sum_{\text{maker trades}} |q_t| (\delta_t + rebatePerNotional \cdot m_t),$$

where δ_t is the distance from mid to fill price (half-spread captured) using the mid at fill.

5.4.13 Step 4: Attribute funding on residual inventory

Compute funding using inventory snapshots at hourly boundaries (or your best available approximation) with oracle notional per the market-maker funding term. This yields a funding PnL series that can be compared to spread and markout series.

5.4.14 Regime intuition: when each term dominates

Market making EV is regime-dependent. Two regimes illustrate why decomposition is not optional:

5.4.15 Low volatility / stable order flow

- μ_{mo} tends to be small (less toxic flow),
- fill rates are decent even with moderate δ ,
- rebates and spread capture can dominate,
- inventory variance is manageable.

5.4.16 High volatility / jumpy microstructure

- μ_{mo} spikes (fills cluster right before moves),
- the same quote width produces worse markouts,
- inventory becomes dangerous (large swings, margin stress),
- the correct response is often to widen, reduce size, or gate quoting.

This is the practical meaning of “toxicity”: it is not a moral label on flow; it is a measurable condition in which your fills have systematically negative expected markouts.

5.4.17 A minimal “edge report” template

A useful daily or hourly report for a market making bot includes:

Line item	Definition (operational)
Gross spread capture	Sum over maker fills of $ q \cdot$ (distance from mid at fill).
Maker rebates / fees	Sum of fee cashflows by trade (rebates positive).
Markout loss (toxicity)	Sum of signed markout PnL at chosen τ horizons.
Inventory PnL	PnL attributable to holding inventory between fills (mark-to-mid or mark-to-oracle consistently).
Funding PnL	Hourly funding cashflows from residual inventory (oracle notional).
Net PnL	Total realized PnL after all above.

The intent is not perfect accounting—different desks allocate inventory drift differently—but rather a stable view that answers: *Are we earning from providing liquidity, or from incidental directional/funding exposure?*

5.4.18 Takeaways

- The sustainable edge comes from **net spread** (realized spread plus maker rebates) exceeding **markout losses** from toxic fills, with inventory risk kept under control.
- A useful EV inequality is the market-maker edge inequality: if markout per fill dominates your net spread plus rebates, you must widen, gate, or reduce size.
- Perps add a distinct term: **funding on residual inventory**. This can help or hurt, and it must be separated in attribution so you do not misdiagnose performance.

With this decomposition in hand, the next step is to turn it into *controls*: inventory-aware quoting, spread widening in volatile regimes,

and hard/soft position limits that keep the strategy alive when microstructure turns hostile.

5.5 Inventory Control and Risk Models

The previous section decomposed market making EV into net spread and rebates minus toxicity, plus inventory and funding effects. Inventory control is the mechanism that turns that decomposition into a survivable strategy: you do not merely “quote both sides”—you continuously adjust where you quote and how wide you quote so that inventory mean-reverts *before* it becomes a liquidation problem or a persistent drag on EV.

On Hyperliquid perps, inventory is risk along three channels at once:

- **Mark-to-market exposure:** your PnL responds to price changes through your net position.
- **Funding exposure:** residual inventory pays or receives hourly funding (oracle-notional-based).
- **Liquidation cliff risk:** if equity falls below maintenance margin, liquidation triggers; if equity drops below **2/3 of maintenance margin** without successful closure via the order book, backstop liquidation occurs via an HLP component strategy.

The last point is the critical operational asymmetry: a market maker can bleed slowly for hours, then fail abruptly when inventory coincides with a fast move and thin liquidity. Inventory control is therefore not optional; it is the core of the strategy.

5.5.1 A state variable: inventory as the control knob

Let Q_t denote your current inventory in base units (positive long, negative short). Inventory evolves as your quotes are hit:

$$Q_{t+\Delta t} = Q_t + \sum_{j \in F(t, t+\Delta t)} s_j q_j,$$

where $s_j = +1$ for buy fills (you add inventory) and $s_j = -1$ for sell fills (you reduce inventory).

Your quoting policy chooses two things continuously:

- **Reservation price** (where you are centered): shift your internal “fair” price up/down to encourage inventory reduction.
- **Spread** (how wide you quote): widen when risk is high (volatility, toxicity, inventory magnitude), tighten when conditions are benign.

This section provides implementable formulas for both, without relying on any single idealized model.

5.5.2 Reservation price: skew to make inventory mean-revert

A clean way to control inventory is to shift the price at which you are willing to buy or sell, relative to a base fair value m_t (mid or microprice from your quoting engine, as developed later in the chapter).

Define a reservation price:

$$r_t = m_t - \phi(Q_t),$$

where $\phi(Q)$ is an increasing function: if you are long ($Q > 0$), then $\phi(Q) > 0$ and $r_t < m_t$, meaning you prefer to sell (you reduce willingness to buy). If you are short ($Q < 0$), then $\phi(Q) < 0$ and $r_t > m_t$, meaning you prefer to buy.

5.5.3 A practical linear skew

The simplest robust choice is linear:

$$\phi(Q_t) = k_{inv} \cdot Q_t,$$

where k_{inv} has units of price per unit inventory. You can interpret k_{inv} as: how many dollars you are willing to concede in price to reduce one unit of inventory.

Then define your quotes around the reservation price:

$$b_t = r_t - \delta_t, \quad a_t = r_t + \delta_t,$$

with δ_t the half-spread you choose.

This automatically produces skew:

- If Q_t becomes **long**, r_t shifts down. Both bid and ask shift down, but the ask becomes more attractive relative to the current mid, increasing sell fill probability and reducing inventory.
- If Q_t becomes **short**, r_t shifts up, making the bid more attractive and increasing buy fills.

5.5.4 Why “skew both sides” is better than moving only one side

Many beginners try to “fix inventory” by only moving the side that reduces it (e.g., for a long inventory, push the ask closer). That often creates unintended behavior:

- Your bid may remain too aggressive, so you continue buying while trying to sell.
- You become predictable: informed takers can lean on the side you kept unchanged.

Shifting the center via r_t is cleaner: you simultaneously become less eager to add inventory and more eager to reduce it.

5.5.5 Spread control: widen when inventory and uncertainty rise

Skew alone is not enough. When risk is high, you also need to widen spreads so that (i) you demand more compensation per fill, and (ii) you reduce fill intensity.

A simple spread function combines three ingredients:

$$\delta_t = \delta_0 + \alpha_\sigma \hat{\sigma}_t + \alpha_Q |Q_t|,$$

where:

- δ_0 is a minimum half-spread (tick constraints set a floor),
- $\hat{\sigma}_t$ is a short-horizon volatility estimate (microstructure-relevant),
- $\alpha_Q|Q_t|$ widens spreads as inventory grows.

This formula is deliberately blunt. Its job is to prevent the worst failure mode: accumulating a large position in the same direction while volatility is rising.

5.5.6 Probabilistic view: controlling fill rates

Spread choices should be thought of as controlling fill probabilities. If you have an empirical model of fill intensity $\lambda(\delta)$ (as in the EV model in the prior section), then widening δ reduces the rate at which you get filled, which mechanically slows inventory accumulation.

A practical implementation uses target fill rates rather than raw spreads:

$$\lambda_t^{target} = \lambda_0 \cdot \exp(-c_\sigma \hat{\sigma}_t - c_Q |Q_t|),$$

and then chooses δ_t so that your observed fill rate matches the target. This approach is robust to different assets and changing market regimes because it adapts to the actual order flow you see.

5.5.7 Inventory bands: soft and hard limits

Model-based skewing helps, but you also need explicit limits because perps have liquidation cliffs.

Define:

- **Soft limit** Q_{soft} : beyond this, you increase skew and widen aggressively.
- **Hard limit** Q_{hard} : beyond this, you stop providing liquidity on the side that increases inventory, and you actively reduce inventory (potentially taking liquidity).

A common, implementable policy:

if $|Q_t| \leq Q_{soft}$: normal quoting; if $Q_{soft} < |Q_t| \leq Q_{hard}$: risk-off quoting; if $|Q_t| > Q_{hard}$: liquidation

5.5.8 Forced de-risk: choose certainty over rebates

When you breach Q_{hard} , your objective changes. You are no longer maximizing spread EV; you are minimizing the probability of hitting liquidation mechanics during a fast move. That often means:

- cancel quotes that increase inventory,
- use more aggressive execution to flatten (even if you pay spread),
- temporarily halt quoting until inventory returns to a safe band.

Because Hyperliquid liquidations proceed via order-book closure attempts first, then backstop via HLP below a deeper threshold, your risk system should aim to avoid ever approaching those states. Inventory controls are most effective when they act *well before* maintenance constraints become tight.

5.5.9 Funding-aware inventory control: inventory has carry

Inventory in perps is special: Q_t has a funding carry term (paid or received hourly, oracle-notional-based). This implies that the “best” inventory is not always zero.

Let f_t be the current hourly funding rate (positive means longs pay). The expected hourly funding PnL from inventory is:

$$\mathbb{E}[\Pi_{fund,t \rightarrow t+1}] \approx -Q_t \cdot f_t \cdot P_t^{oracle}.$$

5.5.10 Funding-adjusted inventory target

If f_t is persistently positive, short inventory earns funding; if persistently negative, long inventory earns funding. You can incorporate this

by defining a funding-driven inventory target:

$$Q_t^* = -\gamma_f \cdot \frac{f_t}{\hat{\sigma}_t^2 + \epsilon},$$

where $\gamma_f > 0$ sets aggressiveness and ϵ stabilizes the ratio.

Interpretation:

- If funding is strongly positive ($f_t > 0$), $Q_t^* < 0$: you allow (small) short bias.
- If volatility rises ($\hat{\sigma}_t^2$ increases), $|Q_t^*|$ shrinks: you do not chase funding when the market is unstable.

Then skew around deviation from target:

$$\phi(Q_t) = k_{inv}(Q_t - Q_t^*).$$

This integrates funding into inventory control without turning the market maker into a directional carry trader. The target should be modest; large funding bets belong in a dedicated carry strategy (as in the earlier section), not in a quoting engine whose mandate is liquidity provision.

5.5.11 Inventory skew explanation (with depth chart visual)

A useful mental model is to imagine that as your inventory becomes increasingly long, you want to (i) pull your bids away from fair value (stop buying), and (ii) move your asks closer to fair value (sell sooner). The same logic applies symmetrically when inventory is short.

5.5.12 A liquidation-aware risk overlay

Skewing and widening are continuous controls. Perp liquidation risk is a discontinuous event. Your inventory system should therefore include a separate overlay that converts current exposure into a “distance-to-trouble” metric and forces de-risking before the cliff becomes plausible.

Even without reproducing the full maintenance margin math, you can build a robust overlay with stress scenarios.

5.5.13 Stress-based distance metric

Choose a set of stress returns $R \in \{-r_1, -r_2, -r_3, +r_1, +r_2, +r_3\}$ and compute the projected equity impact from your current net position. For a perp inventory Q_t at price m_t :

$$\Delta PnL(R) \approx Q_t \cdot (m_t(1 + R) - m_t) = Q_t \cdot m_t \cdot R.$$

Then define:

$$StressLoss = \max_{R \in \mathcal{R}} (-\Delta PnL(R)).$$

Trigger risk-off quoting or forced de-risk if:

$$StressLoss > \eta_{stress} \cdot E_t,$$

where E_t is equity allocated to the strategy and η_{stress} is a conservative fraction (e.g., a few percent). This is intentionally coarse; it is meant to fire early.

5.5.14 Mapping overlay to actions

When the overlay triggers:

- increase α_Q in the spread rule (widen faster),
- increase k_{inv} in the linear skew rule (skew harder),
- reduce quote sizes q ,
- if severe, stop quoting the inventory-increasing side and flatten inventory.

5.5.15 Putting it together: a complete inventory control policy

A practical control stack looks like this:

- **Compute base fair value** m_t (mid/microprice).
- **Compute inventory target** Q_t^* (optional, small) using funding-aware rule the optimal quote-size expression.

- **Set reservation price** $r_t = m_t - k_{inv}(Q_t - Q_t^*)$.
- **Set half-spread** δ_t via the spread rule or fill-rate targeting.
- **Generate quotes** via the reservation-price quoting rule.
- **Apply soft/hard inventory bands** to adjust k_{inv} , δ_t , and quote sizes.
- **Apply liquidation-aware overlay** (stress metric) to force de-risk when needed.

5.5.16 Implementation notes for automated bots

Inventory control is where bots often burn through their operational budgets. Two practical notes:

5.5.17 Update thresholds

As we saw in the order management discussion, you should not update quotes on every small change. Make the control *event-driven*:

- update if $|Q_t - Q_{last}|$ exceeds an inventory threshold,
- update if $\hat{\sigma}_t$ changes materially,
- update if funding changes enough to move Q_t^* or materially change EV.

This reduces unnecessary cancels/modifies while keeping the control loop responsive when it matters.

5.5.18 Separate maker mode from de-risk mode

Keep two execution modes:

- **Maker mode:** post-only/add-liquidity-only as default, sizes tuned for stable regimes.

- **De-risk mode:** prioritize completion (often more aggressive execution) and reduce inventory quickly.

Switching modes based on inventory bands and stress overlay is more robust than trying to continuously interpolate between them.

5.5.19 Takeaways

Inventory control is the bridge from a positive theoretical market making EV to a strategy that survives perps:

- Use a **reservation price shift** to skew quotes so inventory mean-reverts.
- Widen spreads and/or reduce sizes as **inventory magnitude** and **volatility** rise; think in terms of controlling fill rates.
- Enforce **soft and hard limits** and add a **liquidation-aware overlay** to avoid the cliff behavior created by maintenance margin and liquidation mechanics.
- Incorporate funding only as a **small inventory target tilt**, not as a hidden directional bet.

With these controls in place, the next step is to build quoting algorithms that produce stable fair value estimates from L2 order book and trade data, so that your skew and spread policies act on high-quality inputs rather than noisy mids.

5.6 Quoting Algorithms

The prior section developed inventory control rules—reservation price shifts and volatility/inventory-based spread adjustment—that tell you how to move your quotes. This section supplies the missing input: a fair-value estimator that is stable at high frequency, reacts quickly to genuine information, and does not churn orders unnecessarily.

On Hyperliquid, quoting logic must be built around three practical constraints:

- **Block/ordering semantics and cancel priority rules are first-order.** Cancel/replace reliability and post-only behavior are not details; they shape how aggressively you can manage stale quotes.
- **Time-in-force is part of the algorithm.** Post-only/add-liquidity-only (ALO) is the default for rebate/spread capture; IOC is a risk tool for fast inventory reduction; triggers can implement stop-outs.
- **Rate limits are a design constraint.** Practical systems must budget cancels/updates per minute, understand batching behavior, and respect address-based action budgets tied to traded volume.

The goal is to convert real-time L2 order book and trade prints into a robust internal mid m_t and short-horizon volatility estimate $\hat{\sigma}_t$ that drive:

$$r_t = m_t - k_{inv}(Q_t - Q_t^*), \quad b_t = r_t - \delta_t, \quad a_t = r_t + \delta_t,$$

as established previously.

5.6.1 Data inputs: WebSocket-first with snapshot reconciliation

Low-latency quoting requires streaming L2 updates and trades. WebSocket-based market data is recommended, but reconnect logic is a part of the strategy: if you miss updates and continue quoting on a stale book, your markouts will deteriorate rapidly.

A practical data-handling policy:

- 1) Maintain an in-memory L2 book (top N levels) updated by WebSocket deltas.
- 2) On startup and after any reconnect, fetch a fresh snapshot and only resume quoting once: the snapshot is loaded, deltas are applied from a known point, and your derived metrics (microprice, imbalance) pass sanity checks.

- 3) During outages or book inconsistency, switch to a safe mode: widen, reduce size, or pause quoting.

This is not only engineering hygiene; it is part of your edge. The best fair-value model is useless if it is fed corrupted state.

5.6.2 Fair value from L2: microprice and imbalance

The midprice $(a_1 + b_1)/2$ is a weak estimator in fast markets because it ignores depth asymmetry. A better estimator uses the L1 quantities:

$$b_1, a_1 \quad (\text{best bid/ask}), \quad v_1^b, v_1^a \quad (\text{size at best bid/ask}).$$

5.6.3 Microprice (L1 depth-weighted mid)

A standard microprice is:

$$m_t^\mu = \frac{a_1 v_1^b + b_1 v_1^a}{v_1^b + v_1^a}.$$

Intuition: if bid depth is heavy (v_1^b large) and ask depth is light, the microprice moves up toward the ask, reflecting upward pressure.

A useful related statistic is normalized imbalance:

$$I_t^{(1)} = \frac{v_1^b - v_1^a}{v_1^b + v_1^a} \in [-1, 1].$$

5.6.4 Extending to L2: multi-level imbalance

L1 is fast but noisy. A more stable estimator uses multiple levels, down-weighting deeper liquidity:

$$V_t^b = \sum_{i=1}^N w_i v_i^b, \quad V_t^a = \sum_{i=1}^N w_i v_i^a, \quad w_i = \exp(-\eta(i-1)).$$

Then the L2 imbalance is:

$$I_t^{(N)} = \frac{V_t^b - V_t^a}{V_t^b + V_t^a}.$$

You can map imbalance into a price adjustment:

$$m_t = mid_t + \kappa_I \cdot I_t^{(N)} \cdot tickValue,$$

where $tickValue$ is the minimum price increment (or a volatility-scaled unit) and κ_I controls sensitivity.

5.6.5 Practical calibration

Start by setting N small (e.g., 5–15 levels) and η so that w_i halves every few levels. Then estimate κ_I by regressing short-horizon returns on imbalance:

$$\Delta m_{t \rightarrow t+\tau} \approx \beta_I I_t^{(N)} + \varepsilon_t,$$

and set κ_I proportionally to β_I with a strong shrinkage toward zero. In many assets, imbalance is predictive only in bursts; a conservative κ_I avoids overreacting.

5.6.6 Fair value from trades: signed order flow and short-term momentum

Order books can be spoofed or can reshuffle without real intent. Trade prints incorporate commitment: someone crossed the spread. Combining L2 with trades usually yields a more robust m_t .

5.6.7 Signed trade imbalance

Over a rolling window W (seconds), compute signed trade flow:

$$F_t = \sum_{j \in (t-W, t]} s_j \cdot \text{notional}_j,$$

where $s_j = +1$ if buyer-initiated and $s_j = -1$ if seller-initiated. Normalize by total notional to keep scale stable:

$$\tilde{F}_t = \frac{F_t}{\sum_{j \in (t-W, t]} \text{notional}_j + \epsilon}.$$

Then adjust fair value:

$$m_t = m_t^\mu + \kappa_F \cdot \tilde{F}_t \cdot \hat{\sigma}_t \cdot m_t^\mu.$$

This expresses an important principle: order flow is only meaningful relative to current volatility/liquidity. Scaling by $\hat{\sigma}_t$ reduces overreaction in calm periods and increases responsiveness when the market is actually moving.

5.6.8 A blended estimator

A robust default is a convex combination:

$$m_t = \theta m_t^\mu + (1 - \theta) \left(mid_t + \kappa_I I_t^{(N)} \right) + \kappa_F \tilde{F}_t \hat{\sigma}_t m_t,$$

with $\theta \in [0, 1]$. In practice, implement it sequentially: compute a base price from the book, then add the trade-flow adjustment.

5.6.9 Volatility estimation for quoting: microstructure-relevant, not daily

Spread rules in the inventory section require a short-horizon $\hat{\sigma}_t$ that reflects the scale of likely mid moves over the next few seconds/minutes.

5.6.10 EWMA of mid returns

Let m_t be your current fair estimate. Define returns sampled at a fixed interval Δ (e.g., 250ms–1s):

$$r_t = \ln \left(\frac{m_t}{m_{t-\Delta}} \right).$$

Then an EWMA variance:

$$\hat{\sigma}_t^2 = (1 - \lambda) \hat{\sigma}_{t-\Delta}^2 + \lambda r_t^2,$$

with λ chosen so the half-life is on the order of tens of seconds. This estimator is cheap, stable, and sufficient for spread widening.

5.6.11 Jump sensitivity and vol floors

In thin books, a single trade can create a large r_t and temporarily blow up $\hat{\sigma}_t$. Two stabilizers help:

- **Floor:** $\hat{\sigma}_t \leftarrow \max(\hat{\sigma}_t, \sigma_{min})$ to avoid over-tight quotes in calm periods.
- **Cap on update:** clip r_t^2 contribution or use a Huberized return so one print does not freeze quoting for minutes.

5.6.12 Quoting ladders: two-level recipes that trade off edge and risk

A single bid and ask is rarely optimal. A small ladder often improves fill quality and inventory control.

5.6.13 Two-level ladder

Define for each side two quotes:

$$\begin{aligned} b_t^{(1)} &= r_t - \delta_t^{(1)}, & a_t^{(1)} &= r_t + \delta_t^{(1)}, \\ b_t^{(2)} &= r_t - \delta_t^{(2)}, & a_t^{(2)} &= r_t + \delta_t^{(2)}, \end{aligned}$$

with $\delta_t^{(2)} > \delta_t^{(1)}$.

Interpretation:

- Level (1) competes near top-of-book for consistent fills.
- Level (2) is a backstop that fills during dips/spikes, often improving average entry/exit for inventory reduction without constantly chasing.

A volatility-adaptive spacing is:

$$\delta_t^{(1)} = \max(\delta_0, c_1 \hat{\sigma}_t m_t), \quad \delta_t^{(2)} = \max(2\delta_0, c_2 \hat{\sigma}_t m_t),$$

with $c_2 > c_1$.

5.6.14 Size allocation across levels

Set size larger at the safer level in risky regimes:

$$q_t^{(1)} = q_0 \cdot \exp(-\alpha \hat{\sigma}_t), \quad q_t^{(2)} = q_0 - q_t^{(1)},$$

so when volatility rises you reduce your most aggressive size and shift activity deeper.

5.6.15 Inventory-linked quote selection: fade-to-flat as a mode switch

The inventory section gave continuous skew and spread widening. In practice, you often want a discrete switch when inventory breaches a threshold: stop trying to earn spread symmetrically and focus on flattening.

Define a threshold Q_{fade} .

5.6.16 Normal mode ($|Q_t| \leq Q_{fade}$)

- Quote both sides with ALO as default.
- Apply reservation-price skew and volatility-based widening smoothly.

5.6.17 Fade-to-flat mode ($|Q_t| > Q_{fade}$)

- Move the inventory-reducing side closer (increase fill probability).
- Pull the inventory-increasing side away or cancel it entirely.
- Use opportunistic IOC (or other aggressive execution) when the book offers a favorable moment to reduce inventory quickly.

This discrete logic is easier to reason about than continuously escalating parameters, and it reduces the chance you slowly drift into an unsafe inventory region.

5.6.18 Order management on Hyperliquid: making quotes safe under cancels and limits

Even a good fair value estimate can lose money if stale quotes are picked off. Hyperliquid's cancel priority rules and block/ordering semantics affect how quickly you can remove risk. That translates into one practical design principle:

Treat quote updates as a scarce resource; update only when the expected cost of staleness exceeds the expected cost of consuming an action.

5.6.19 Thresholded quote updates

Maintain your last posted quotes (b_t^{post}, a_t^{post}) . Update only if one of these triggers fires:

- 1) Fair value moved: $|m_t - m^{post}| > \Delta_m$.
- 2) Volatility moved: $|\hat{\sigma}_t - \hat{\sigma}^{post}| > \Delta_\sigma$.
- 3) Inventory moved: $|Q_t - Q^{post}| > \Delta_Q$.
- 4) Queue decay: your queue position is likely deteriorating (heuristic), or time since last refresh exceeds a timeout.

This preserves action budget and reduces self-induced churn.

5.6.20 Post-only default with a safety valve

Use post-only/add-liquidity-only (ALO) for your normal ladder to capture spread and rebates. But implement a safety valve: if stale risk rises (sudden $\hat{\sigma}_t$ jump, large trade flow spike, inventory breach), cancel and switch to a more urgent order type to reduce exposure. This is consistent with the earlier EV decomposition: rebates help, but they are never worth getting run over.

5.6.21 Batching and address-based action budgets

Batching order operations can reduce IP-weighted request usage, but your strategy must still respect address-based action consumption. The practical implication is that quote ladders and refresh logic should be designed to keep a stable number of resting orders and avoid constant cancel/replace loops.

A simple budget rule:

$$\text{ExpectedActionsPerMin} \approx \text{numQuotes} \times \text{refreshRate} \leq \text{budget},$$

and if the inequality fails, you widen thresholds (Δ_m, Δ_σ), reduce ladder depth, or increase refresh intervals.

5.6.22 A reference quoting loop (pseudocode)

The following pseudocode ties the components together: L2+trade fair value, EWMA volatility, inventory-skewed reservation price, two-level ladder, and thresholded updates with a fade-to-flat mode.

```
def quoting_loop(state, l2, trades, risk, params):
    # 1) Build fair value from L2
    b1, a1, vb1, va1 = l2.best_bid(), l2.best_ask(), l2.best_bid_size(), l2.best_ask_size()

    mid = 0.5 * (b1 + a1)

    # Microprice (L1)
    micro = (a1 * vb1 + b1 * va1) / max(vb1 + va1, params.eps)

    # L2 weighted imbalance
    Vb, Va = l2.weighted_depth(params.N, params.eta) # returns (Vb, Va)
    imb = (Vb - Va) / max(Vb + Va, params.eps)

    base = params.theta * micro + (1 - params.theta) * (mid + params.kl * imb *
        params.tick)

    # 2) Trade flow adjustment
    flow = trades.signed_flow(window_sec=params.flow_window) # normalized to [-1, 1]
    m = base + params.kF * flow * state.sigma * base

    # 3) Update short-horizon volatility (EWMA on log returns)
    r = np.log(max(m, params.eps) / max(state.m_prev, params.eps))
    state.sigma2 = (1 - params.lam) * state.sigma2 + params.lam * min(r * r, params.
        r2_cap)
    state.sigma = max(np.sqrt, params.sigma_floor)
    state.m_prev = m
```

```
# 4) Inventory target tilt from funding (optional; small)
Q_star = -params.gamma_f * risk.funding_rate() / (state.sigma2 + params.eps)

# 5) Reservation price and spreads
Q = state.inventory
rsv = m - params.kinv * (Q - Q_star)

d1 = max(params.d0, params.c1 * state.sigma * m)
d2 = max(2 * params.d0, params.c2 * state.sigma * m)

# 6) Fade-to-flat mode
fade = abs(Q) > params.Q_fade
if fade:
# Encourage inventory reduction: tighten reducing side, pull away increasing side
if Q > 0: # long: prefer sells
a1p, a2p = rsv + 0.6 * d1, rsv + 0.8 * d2
b1p, b2p = rsv - 1.6 * d1, rsv - 2.0 * d2
else: # short: prefer buys
b1p, b2p = rsv - 0.6 * d1, rsv - 0.8 * d2
a1p, a2p = rsv + 1.6 * d1, rsv + 2.0 * d2
else:
b1p, b2p = rsv - d1, rsv - d2
a1p, a2p = rsv + d1, rsv + d2

# 7) Thresholded updates to respect action budget
if should_update_quotes(state, m, state.sigma, Q, params):
# Post-only as default; IOC used elsewhere for urgent inventory reductions
place_or_modify_quotes_post_only(
bids=[(b1p, params.q1), (b2p, params.q2)],
asks=[(a1p, params.q1), (a2p, params.q2)],
)
state.snapshot(m=m, sigma=state.sigma, Q=Q)
```

This loop intentionally separates:

- **estimation** (L2 microprice, imbalance, trade flow, volatility),
- **control** (reservation price, spreads, fade-to-flat),
- **operations** (thresholded updates to conserve actions).

5.6.23 Practical calibration workflow

To tune the system without overfitting:

- 1) Start with **microprice only** (the L1 microprice) and a conservative spread rule.
- 2) Add **L2 imbalance** (the L2 order-book imbalance) with small κ_I and verify it improves markout on a holdout period.

- 3) Add **trade flow** adjustment only if it reduces negative mark-outs during fast conditions.
- 4) Calibrate thresholds ($\Delta_m, \Delta_\sigma, \Delta_Q$) to fit your **action budget** first; then optimize performance within that constraint.
- 5) Evaluate with the attribution metrics from the EV decomposition section: net spread + rebates should dominate markout losses in the regimes you target.

5.6.24 Takeaways

High-quality quoting on Hyperliquid perps is a fusion of microstructure estimation and operational discipline:

- Use **microprice and L2 imbalance** to estimate fair value more intelligently than the raw mid.
- Use **trade flow** as a secondary adjustment to detect genuine pressure versus cosmetic book movement.
- Estimate **short-horizon volatility** for spread scaling and safety gating.
- Design quote updates around **cancel/replace reliability** and **rate limits**; treat updates as scarce actions.
- Combine post-only quoting for edge capture with a clear, explicit **fade-to-flat** mode and **IOC** safety valve for inventory reduction.

With a fair-value engine and disciplined order management, your inventory control rules become precise rather than reactive, which is what ultimately stabilizes market making EV across regimes.

5.7 Protocol Liquidity and HLP Vaults

The previous sections built a bottom-up view of liquidity provision: estimate fair value from L2 and trades, quote with inventory-aware skew, and measure edge by decomposing spread, rebates, toxicity, inventory

drift, and funding. Hyperliquid adds a top-down liquidity layer: the *Hyperliquid Liquidity Provider (HLP)* vault, a protocol vault that supplies liquidity through multiple strategies and also performs liquidations via a component strategy. For an on-chain trader, HLP is both (i) a large participant that shapes the microstructure you trade against, and (ii) an investable vehicle that offers “liquidity beta” without building an execution stack.

This section treats HLP as a *protocol-native market making portfolio* and answers three practical questions:

1. How does HLP affect observed spreads and depth?
2. What is HLP’s role in liquidation handling, and what risk does that create for depositors?
3. When does it make sense to allocate to HLP versus running your own quoting strategies?

5.7.1 What HLP is (economically)

From an economic standpoint, HLP is a pooled balance sheet that:

- provides liquidity via multiple market-making strategies,
- accrues a portion of trading fees (per docs),
- absorbs liquidation flow via a liquidator vault component strategy when book liquidation is insufficient and the distressed account falls further.

If you deposit into HLP, your PnL is *pro-rata*: depositors share gains and losses in proportion to their shares. This is the same principle as a hedge fund or a vault: you are not earning a fixed yield; you are taking a stake in a strategy.

Two operational constraints matter immediately:

- HLP has a **4-day deposit lock-up**: withdrawal is allowed 4 days after the most recent deposit.

- User vaults typically have a **1-day lock-up** and a **10% performance fee to the leader**; protocol vaults (including HLP) do *not* have that leader fee.

5.7.2 HLP as “liquidity beta”: mapping to the EV decomposition

In the market making decomposition, a quoting strategy’s expected value is driven by:

$$\Pi \approx \Pi_{spread} + \Pi_{fees} - \Pi_{tox} + \Pi_{inv} + \Pi_{fund}.$$

You can interpret HLP as holding a diversified, protocol-operated version of these exposures:

- **Spread and rebate capture:** HLP participates as a maker across markets; this tends to push spreads tighter than they would be with only private makers.
- **Toxic flow and inventory drawdowns:** HLP is exposed when informed takers arrive or when volatility spikes. This is why HLP returns are not monotone in volume: high volume can be good (more rebates/spread) or bad (more toxic flow).
- **Funding coupling:** because HLP’s strategies may carry net inventory at times, they can pay/receive funding; this term can help or hurt depending on regimes.
- **Liquidation absorption:** HLP’s liquidator component strategy takes the other side of certain liquidation events when normal book liquidation fails. This is an additional return/risk stream not present in a plain market maker.

The key is that HLP returns are not simply “fees.” They are the net result of all these components, which can be positively or negatively correlated with market stress.

5.7.3 Impact on liquidity depth and organic spread contraction

When a large participant provides resting liquidity systematically, the order book typically becomes:

- **deeper near the mid** in normal conditions,
- **more stable** across small shocks (because there is always some replenishment),
- **tighter in spread** because competition among liquidity providers increases.

This affects your strategy selection:

For takers (directional entries, hedges, arbitrage completion). Deeper books and tighter spreads reduce slippage and improve fill reliability. That directly increases the feasibility of the basis and arbitrage strategies discussed earlier.

For private makers. Tighter spreads compress gross edge. Your survival depends more on:

- fee tiering and maker rebates,
- better fair value estimation (microprice/flow),
- inventory control discipline,
- niche markets/times when HLP is less aggressive or when your specialization yields better markout outcomes.

A practical implication is that many private makers do *better* by focusing on microstructure niches (e.g., particular assets or times) rather than trying to be a universal top-of-book competitor against HLP.

5.7.4 HLP and liquidation handling: why it matters for both traders and depositors

Hyperliquid liquidations start by attempting to close positions through the order book using market orders. If equity falls below **2/3 of main-**

tenance margin without successful closure through the book, *back-stop liquidation* occurs via a liquidator vault strategy that is a component of HLP.

This design has two important microstructural consequences:

5.7.5 1) Liquidity continuity in stress

In severe moves, order books can gap or thin out. A protocol-native liquidator provides a mechanism to continue closing risk when normal liquidity is inadequate. This reduces systemic tail risk for the venue, which can indirectly benefit all participants by lowering the probability of chaotic cascades.

5.7.6 2) A stress-linked risk premium for HLP depositors

Liquidation absorption is not free. If HLP is taking the other side of distressed flow, it can experience:

- sudden inventory acquisition at unfavorable prices,
- high volatility markouts,
- concentrated exposures during liquidation waves.

In calm markets, this component may contribute positively (capturing a premium for providing backstop liquidity). In crisis regimes, it can produce drawdowns.

For a depositor, the correct mental model is:

HLP may earn a premium for being willing to intermediate stress events, but it also bears stress-event risk.

5.7.7 Lock-ups, correlation, and portfolio design

Because HLP has a 4-day lock-up, it is not a tactical “in/out” instrument. That constraint changes optimal usage:

5.7.8 HLP as a strategic sleeve

A systematic trader can treat HLP as a strategic allocation akin to:

- an exposure to exchange-wide market making profitability,
- plus stress-event risk through liquidation handling.

This can complement strategies whose returns are less tied to exchange-wide volatility regimes (e.g., certain basis or directional systems with explicit risk limits).

5.7.9 Drawdown clustering

Liquidity provision strategies often suffer when volatility spikes: toxicity rises, inventory becomes harder to manage, and liquidation events cluster. If you also run your own market making, you should assume nontrivial correlation with HLP during stress.

A practical portfolio guideline:

- Avoid doubling your exposure to the same regime by allocating heavily to HLP *and* running a similarly exposed private maker book on the same assets.
- If you do both, deliberately diversify: different assets, different quoting horizons, or explicit hedges that reduce the shared tail.

5.7.10 HLP versus running your own maker strategy: a decision framework

The choice is not purely expected return. It is expected return *net of operational complexity* and *net of constraints*.

5.7.11 When HLP is attractive

HLP tends to be attractive if:

- you want exposure to liquidity provision without building and maintaining a low-latency quoting stack,
- you do not have a clear advantage in fair value estimation, queue tactics, or risk overlays,
- you prefer pro-rata exposure to a diversified set of protocol strategies,
- you can tolerate the 4-day lock-up and potential stress draw-downs.

5.7.12 When proprietary market making is attractive

Running your own maker strategy tends to be attractive if:

- you can sustain a data pipeline (WebSocket L2 + trades), robust reconnect/snapshot reconciliation, and strict action budgeting,
- you have demonstrably better markout outcomes in certain regimes or assets,
- you need faster capital mobility than a 4-day lock-up allows,
- you want tight control over inventory, funding bias, and risk caps that a pooled vault cannot tailor to your preferences.

5.7.13 A hybrid approach

Many professional systems use a hybrid:

- Allocate a portion of capital to HLP as a baseline liquidity exposure.
- Run separate strategies in the master account (directional, funding carry, basis/arbitrage, or specialized making).
- Manage correlation explicitly: if volatility rises and your private maker reduces risk, you should assume HLP may simultaneously face a tougher environment; size accordingly.

5.7.14 How HLP shapes your execution costs (practical implications)

Even if you never deposit in HLP, its presence affects your performance:

- **Directional execution:** tighter spreads reduce entry/exit friction; your signal-to-cost inequality becomes easier to satisfy.
- **Carry and basis trades:** better depth improves hedge completion, reducing sequencing risk and lowering the penalty you must charge for Δt .
- **Arbitrage:** more stable top-of-book liquidity increases the success rate of fast hedges, but compressed spreads can also reduce the gross mispricing available.

So HLP is part of your *environment*. Treat it like a structural factor similar to fee schedules and funding rules: it changes what kinds of edges are available and where competition is fiercest.

5.7.15 A concise checklist for depositors

If you consider depositing into HLP, the due diligence questions are straightforward:

Question	Why it matters
Can I tolerate a 4-day lock-up?	You cannot quickly de-risk during fast regime shifts; treat allocation as strategic.
How does it behave in high volatility?	Liquidity strategies often experience draw-downs when toxicity and liquidation events cluster.
Am I already exposed to similar risks?	Private market making, liquidation-sensitive positions, and HLP can all draw down together.
Do I need capital mobility for other strategies?	Arbitrage and carry can require rapid reallocation; lock-up can be costly in opportunity terms.

5.7.16 Takeaways

HLP is best understood as protocol-native liquidity supply with a distinct return profile:

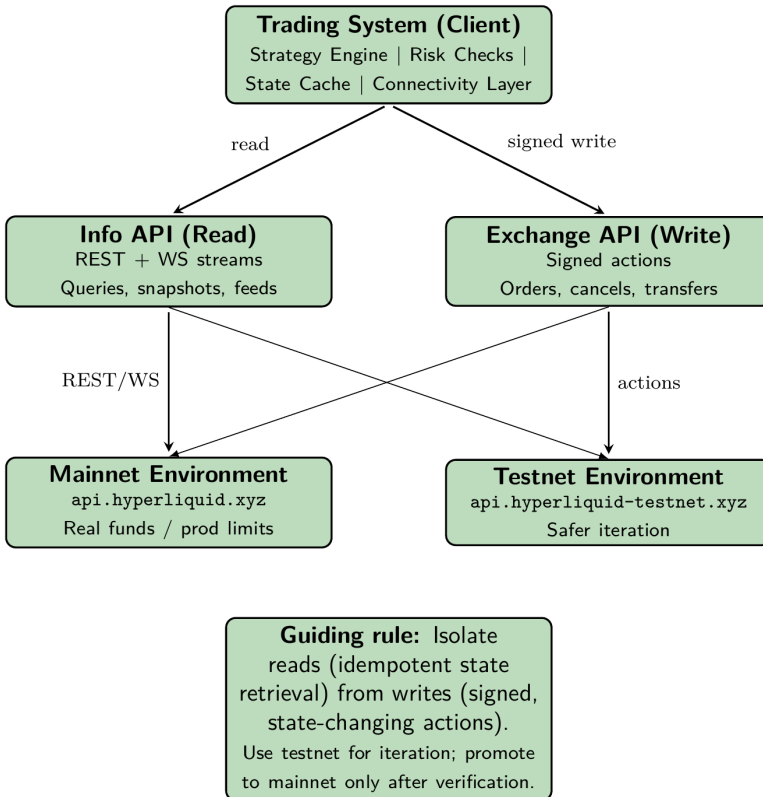
- It provides liquidity through multiple strategies and performs liquidations via a component strategy, which supports venue stability but introduces stress-linked risk for depositors.
- By adding systematic depth, HLP tends to tighten spreads and improve execution quality for takers, while increasing competition for private makers.
- Depositing in HLP is a pro-rata exposure with a **4-day lock-up**; user vaults differ in lock-up (typically 1 day) and have a **10% leader performance fee**, while protocol vaults do not.
- For systematic traders, HLP can be used as a “liquidity beta” sleeve alongside separate directional, funding, and basis strategies, provided correlation and drawdown clustering are managed deliberately.

Chapter 6

API Integration and Exchange Connectivity

A brilliant trading strategy on paper means nothing if it stumbles during execution. In the hyper-competitive arena of on-chain perpetuals, your trading infrastructure must be as razor-sharp as your alpha. This chapter bridges the gap between quantitative theory and operational reality, dissecting the Hyperliquid API architecture to transform your algorithms into living, breathing trading systems capable of handling sub-millisecond on-chain volatility and deterministic execution.

6.1 API Architecture and Environments



Hyperliquid connectivity is easiest to reason about if you impose a strict boundary inside your trading system: *reads* fetch and stream state; *writes* change state. That separation is not just aesthetic. It reduces the chance that an execution bug (placing/canceling/modifying orders) contaminates your market data path, and it makes testing safer because you can replay read traffic without ever producing a trade.

6.1.1 Two API surfaces: Info (read) versus Exchange (write)

Hyperliquid's HTTP interface is organized around two conceptual endpoints:

- **Info API (read-only):** queries for market and account state. These requests should be treated as idempotent: repeating them does not change exchange state.
- **Exchange API (write):** signed actions that mutate state: placing orders, canceling orders, and related account actions. These are *not* idempotent unless you add your own idempotency layer (discussed later in the chapter via client order IDs and reconciliation patterns).

Operationally, you should treat these as two different subsystems with different reliability and security requirements:

- The **read path** cares most about *continuity* (reconnect behavior), *consistency* (snapshot + delta correctness), and *latency* (fresh quotes and positions).
- The **write path** cares most about *authentication* (signatures), *nonce discipline* (to avoid rejected actions under concurrency), and *risk controls* (position limits, reduce-only constraints, kill switches).

A practical architecture is to run the read path as a service that builds a local mirror (prices, L2, positions, open orders), while the write path consumes only that internal mirror plus strategy intent. This makes it possible to “freeze” execution while still observing the market, or to degrade gracefully under partial outages.

6.1.2 REST versus WebSocket: what each transport is for

Hyperliquid exposes both REST and WebSocket (WS) interfaces. Both can deliver read data; writes are typically sent as signed actions (and can also be submitted via a WS post wrapper, covered later).

REST: point-in-time queries.

- **Bootstrapping state** on startup (e.g., initial metadata, a clean starting snapshot of positions/open orders).
- **Spot checks** and **reconciliation probes** (e.g., querying order status when your client suspects a mismatch).
- **Low-frequency analytics** where streaming is unnecessary.

REST's weakness is that it does not inherently provide continuity. If you poll, you must choose a polling frequency and accept that you can miss short-lived events or observe state changes late.

WebSocket: continuous streams and low-latency updates.

WS is best for:

- **Market data streams** (e.g., L2 book updates, trades, best bid/offer).
- **User streams** (e.g., order updates, fills, ledger/funding updates).
- **Building deterministic local mirrors** that update as events arrive.

WS's weakness is that it can disconnect, and your system must be able to detect gaps and re-establish correct state (snapshot + resubscribe + reconcile). Hyperliquid also enforces heartbeat expectations: the server closes a connection if it hasn't sent any message to the client in the last 60 seconds; clients can send { "method": "ping" } and should expect { "channel": "pong" }.

6.1.3 Environment selection: mainnet versus testnet

Hyperliquid provides separate base URLs for mainnet and testnet. Treat them as distinct exchanges: separate latency characteristics, separate state, separate operational risk.

Mainnet

WS: wss://api.hyperliquid.xyz/ws

Testnet

WS: wss://api.hyperliquid-testnet.xyz/ws

REST: https://api.hyperliquid.xyz

REST: https://api.hyperliquid-testnet.xyz

Use testnet for:

- validating that your client can **sign actions** correctly and receive expected acknowledgements;
- verifying **nonce handling** under parallel order placement/cancel bursts;
- hardening your **WS reconnect logic** (including resubscribe storms and snapshot handling);
- exercising **error paths** intentionally (tick-size violations, size/notional minimums, reduce-only rejects, etc.).

Promote to mainnet only when your system can demonstrate two capabilities end-to-end:

1. **State continuity:** after forced disconnects and reconnects, your local view of positions and open orders matches the exchange.
2. **Execution safety:** a bug cannot cascade into repeated, state-changing writes (e.g., infinite retries that spam signed actions).

6.1.4 A safe client mental model: the “read mirror” and the “write actuator”

For on-chain perps, the most common integration failures are not “couldn’t fetch data” but rather *inconsistent state* plus *unsafe retries*. A robust design uses two internal components:

- **Read mirror:** consumes REST snapshots and WS streams to maintain a local, queryable state store (prices, book, positions, margins, open orders, recent fills). This component should be restartable and replayable.

- **Write actuator:** takes decisions from the strategy engine and transforms them into signed actions. It enforces pre-trade checks (max position, max order size, reduce-only policy, order rate limits) and post-trade verification (did the order rest, fill, or reject?).

The key is that the actuator should not “trust” that submitting an action implies it took effect. Instead, it should treat every write as *asynchronous*: you submit, receive an acknowledgement, and then confirm the resulting state via user streams and/or targeted Info queries. This lets you survive transient network errors and intermittent WS disconnects without accidentally doubling orders.

6.1.5 Practical endpoints and transports to implement first

A minimal-but-production-oriented connectivity stack usually implements the following in order:

1. **REST Info queries for bootstrap:** fetch enough state to start safely (e.g., current positions and open orders for the user address).
2. **WS subscriptions for continuous updates:** at minimum, one market stream needed for quoting/alpha and one user stream for execution feedback (orders/fills).
3. **Write actions with signature plumbing:** order placement and cancel, with strict retry rules.
4. **A health layer:** WS ping/pong, reconnect backoff, and a “desync suspected” flag that disables new risk-increasing orders until reconciliation passes.

In code, reflect this structure directly: keep separate modules (or services) for `info_client` and `exchange_client`; keep different configuration keys for each environment; and route all outbound writes through a single concurrency-controlled queue. Those design choices make later chapters—nonces, wallet delegation, and deterministic WS book building—far easier to implement correctly.

6.2 Cryptographic Authentication

The Exchange API accepts *state-changing* requests (placing and canceling orders, and related actions) only if they are authenticated by an L1 ECDSA signature. In practice, this means your trading system must prove: (i) *which signer* authorized the action, and (ii) *exactly what bytes* they authorized. If either is ambiguous, you will see intermittent failures that are hard to reproduce, or worse, you will build a system that signs something different from what you think it signed.

6.2.1 What “ECDSA authentication” means in this context

At a high level, ECDSA signing uses a private key to produce a signature over a message digest. The exchange can then verify the signature using only the corresponding public address. Two properties make this ideal for automated trading:

- **The private key never needs to leave your system.** You transmit only the signature and the unsigned payload.
- **The signature is bound to the payload.** If the payload changes (even a single byte), the signature no longer verifies.

From an integrator’s perspective, it helps to keep a concrete mental model of the signature object:

- An ECDSA signature is typically represented as (r, s, v) (or equivalently a 65-byte signature where r and s are 32 bytes each, plus a 1-byte recovery identifier v).
- Verification consists of reconstructing a digest $H(m)$ from the payload m , then checking that the signature corresponds to the claimed signer address when recovered against that digest.

The practical implication is that *the digest construction is part of the protocol*. If you serialize fields differently than the exchange expects, you are signing the “wrong” message even though the data *looks* the same at the JSON level.

6.2.2 Why SDK parity matters (and why signature bugs look random)

Hyperliquid documentation strongly recommends using an SDK for signature generation because subtle serialization mismatches are common failure modes. These bugs often appear random because a subset of payloads accidentally serialize the same way across two encoders, while others do not.

Two signing schemes are referenced in the docs (Python SDK methods):

- `sign_l1_action`
- `sign_user_signed_action`

Confusing these is a common integration error: the exchange will recover an unexpected signer address, or reject the action because the signature does not validate under the expected scheme. Treat the signing method as part of your API contract, not an implementation detail.

6.2.3 Threat model: what you are defending against

Before you decide *how* to sign, be explicit about what can go wrong:

- **Key exfiltration:** an attacker obtains your private key (or an API agent key) and can sign arbitrary actions.
- **Payload substitution:** malware or a compromised dependency causes your system to sign a different payload than the one your strategy intended (e.g., different asset, size, price, or `reduceOnly` flag).
- **Replay and confusion attacks:** signed messages are reused, reordered, or interpreted under a different domain/meaning than intended.
- **Operational leakage:** private keys appear in logs, crash dumps, shell history, CI artifacts, or analytics events.

Good signing design reduces these risks by (i) keeping signing isolated, (ii) using domain-separated, typed message formats where possible, and (iii) minimizing the blast radius of any single key.

6.2.4 Typed structured signing versus legacy signing

In Ethereum ecosystems, the canonical framework for readable, domain-separated off-chain signatures is **EIP-712**: typed structured data hashing and signing. The key benefit is that the signature is not just over an opaque blob; it is over a structured message with an explicit type schema and domain separator. This domain separation is a defense against signing the “same bytes” in a different context.

By contrast, legacy flows such as `personal_sign/eth_sign` are easier to misuse in automated systems because they often sign arbitrary text or byte strings with fewer structural guardrails. From a trading-bot risk perspective, the question is not philosophical: it is about reducing the probability that a UI string, a JSON normalization change, or a library update alters the signed digest without triggering an immediate failure in your pre-trade checks.

OpenZeppelin’s EIP712 guidance emphasizes careful encoding (the familiar `abi.encode + keccak256` patterns): the encoder is part of the security boundary. That is precisely why SDK parity matters; “nearly identical” encodings are not identical at the byte level.

6.2.5 Signing pipeline: isolate responsibilities

A production-grade trading system should treat signing as a small, heavily audited subsystem with a narrow interface. A clean separation looks like this:

- 1. **Strategy intent**: “I want to place/cancel these orders.”
- 2. **Canonical payload builder**: construct the action payload in the exact field order/format expected by the exchange.
- 3. **Digest construction**: compute the hash according to the required signing scheme.

- **4. Signature creation:** produce (r, s, v) using the private key in a hardened signer component.
- **5. Local verification:** recover the signer address from the signature and digest, and assert it matches the expected address *before* sending the action.
- **6. Submission:** send payload + signature to the Exchange API.

The “local verification” step is a powerful sanity check: it catches scheme mismatches and malformed payload encodings earlier, with better error messages, and without burning rate limits on invalid actions.

6.2.6 Sequence of a signed action

6.2.7 Key management: how to sign without exposing private keys

Signing is only as secure as key custody. For trading systems, the goal is not “perfect” security; it is to reduce the probability and impact of compromise while keeping the system operable.

Principle 1: never log secrets.

This includes private keys, seed phrases, raw signatures for sensitive payloads, and any serialized blob that could be replayed. Configure your logger to redact values by key name and to disable debug logging in production.

Principle 2: isolate the signing surface.

Treat the signer as a separate component (often a separate process, container, or host) that exposes one function: “sign this canonical payload.” Do not let it perform networking, strategy decisions, or state storage. Fewer dependencies means fewer supply-chain risks.

Principle 3: minimize blast radius.

Even if a key is compromised, you want it to be the *least powerful* key that still enables the strategy. In Hyperliquid, this naturally leads to

API wallets (agent wallets) as an operational pattern; the mechanics and nonce implications are covered in the next section.

Principle 4: make signing deterministic and testable.

Deterministic here means: for a given payload under a given signing scheme, the bytes hashed and the signature produced are reproducible (modulo ECDSA’s internal randomness, depending on implementation). Your test suite should include fixtures that:

- build a payload,
- sign it via the same code path used in production,
- locally recover the signer,
- and assert it matches the expected address.

6.2.8 Signature failure taxonomy (what breaks most often)

The table below lists common causes of signature failures and the corresponding engineering response. These failure modes are exactly why the official guidance is to use an SDK rather than “rolling your own” encoder.

Failure mode	What it looks like / mitigation
Wrong signing scheme (sign_ll_action vs sign_user_signed_action)	Exchange recovers an unexpected signer or rejects signature validation. Lock the scheme per action type; add local recover-checks.
Field order / serialization mismatch	Some payloads sign/verify, others fail. Use SDK canonicalization; avoid “helpful” JSON transformations.
Numeric formatting quirks	Values that are numerically equal may serialize differently (e.g., trailing zeros). Ensure the exact encoding matches the SDK/exchange expectation.
Address casing differences	Recovered signer mismatch or inconsistent hashing if addresses are not normalized. Normalize addresses consistently (docs recommend lowercasing).
Local verify differs from exchange verify	Your local digest construction is not identical to the exchange’s. Verify with the same scheme and encoding as the SDK; add golden test vectors.

6.2.9 Implementation discipline: a minimal signing interface

Even if you use an SDK, it is worth constraining your internal API so that the rest of your system cannot bypass safety checks.

```
class Signer:
    def expected_address(self) -> str:
        """Return the lowercased expected signer address."""
        raise NotImplementedError

    def sign_action(self, action_payload: dict) -> bytes:
        """
        Produce an L1 ECDSA signature over the canonical encoding of the payload.
        The concrete implementation should use the SDK signing routine.
        """
        raise NotImplementedError

    def prepare_signed_request(signer: Signer, payload: dict) -> dict:
        # 1) Canonicalize payload exactly as required by the signing scheme (SDK
            responsibility).
        # 2) Produce signature.
        sig = signer.sign_action(payload)

        # 3) Locally recover/check signer (implementation-specific; keep this step mandatory).
        # If recovery!= signer.expected_address(), abort and alert.
        #
        # 4) Return transport object for Exchange API submission.
        return {"payload": payload, "signature": sig}
```

The important takeaway is architectural: the rest of your trading system should not need to know anything about hashing, field ordering, or signature serialization. It should hand a structured action to a hardened signer module and receive a signature (or an explicit error).

6.2.10 How this connects to the rest of the chapter

From the prior section, you already have a clean separation between Info (read) and Exchange (write). Cryptographic authentication is what turns “write” into “authorized write.” Once signing works, the next operational bottleneck is *coordination*: how to submit many signed actions concurrently without nonce collisions and without granting full-power keys to every process. That is the motivation for the next section on nonces and API wallets (agent wallets).

6.3 Nonces and API Wallets

Once you can sign Exchange API actions reliably, the next failure mode is rarely cryptographic. It is *coordination*: multiple processes (or threads) attempting to submit signed actions at the same time, sometimes for multiple subaccounts, all while keeping keys safe. Hyperliquid addresses this operational reality with two intertwined mechanisms:

- **Nonces** to make each signed action unique and to impose a well-defined ordering constraint.
- **API wallets (agent wallets)** to delegate trading authority without running your main wallet key on an always-on server.

This section explains how to reason about both, and how to design a bot that is fast, concurrent, and difficult to accidentally compromise.

6.3.1 API wallets (agent wallets): delegation as a security primitive

An **API wallet (agent wallet)** is a separate keypair that you authorize to sign actions on behalf of a *master account* (and potentially subaccounts). The key operational idea is simple: your always-on trading infrastructure signs with the agent key, not with the master wallet's private key.

Two details from the docs are easy to miss and matter immediately:

- **Agent wallets can sign actions for the master or subaccounts once approved.** This is the intended way to run automated systems.
- **Queries still require the actual user address.** Even if an agent wallet signs actions, account state queries (Info API) still use the master account address (the user address), not the agent address.

If you blur these roles, you will end up debugging “missing positions” or “no open orders” simply because the read path is pointed at the wrong address.

Operational best practice: treat keys by role.

A clean key layout usually looks like:

- **Master wallet key:** kept offline or in a highly restricted environment; used to approve agent wallets and manage account-level administration.
- **Agent wallet key(s):** stored in production signing infrastructure; used only to sign Exchange actions.

This gives you a straightforward blast-radius story: a compromised agent can trade (bad), but it should not automatically imply full administrative control of the master key (worse). In later chapters you can tighten this further with risk limits, reduce-only logic, and kill switches, but key separation is the first layer.

6.3.2 What a nonce is, and why exchanges insist on it

A **nonce** is a number included in the signed message that makes each action unique. If an attacker records a signed action and replays it later, a well-designed nonce rule causes the replay to be rejected.

Nonces also help exchanges manage *ordering*. If your system submits “cancel” and “place” actions close together, the exchange needs a consistent way to decide whether a later action should override an earlier one, and it needs a way to discard duplicates.

For trading systems, there are two practical consequences:

1. You must ensure **each action uses a nonce that the exchange will accept**.
2. You must ensure **your own concurrency does not cause nonce collisions**.

6.3.3 Hyperliquid's nonce rules (what is actually enforced)

Hyperliquid tracks nonces **per signer private key**. Concretely:

- If you sign directly with your main wallet, nonces are tracked under that wallet address.
- If you sign with an agent wallet, nonces are tracked under the *agent address*.

This is the first concurrency trap: **separate subaccounts signed by the same agent share nonce tracking**. If two independent workers share one agent key and both emit actions concurrently, they are competing for the same nonce space.

Hyperliquid's acceptance rule is designed for throughput and mild out-of-order arrival:

- The exchange stores the **100 highest nonces per signer**.
- A new nonce must be **greater than the smallest nonce in that stored set** and must be **unused**.
- The nonce must fall within a relative time window: $T - 2$ **days** to $T + 1$ **day** (where T is the current block time on the relevant reference).

This is not the strict “nonce must be exactly +1” model some systems use. Hyperliquid tolerates a degree of out-of-order submission, as long as you stay within the top-100 “high watermark” set and never reuse a nonce.

Intuition for the top-100 rule.

Think of the exchange maintaining a sliding buffer of recently used high nonces. You can submit nonces slightly out of order (useful if you have bursts, retries, or multiple in-flight actions), but you cannot:

- go too far backward (below the current acceptance floor), or
- reuse a value that is already in the recent set.

6.3.4 Designing a nonce allocator that works under concurrency

Because nonces are per signer key, you want a *single* allocator per key that hands out unique numbers across all threads/processes that can sign with that key. There are two common patterns.

Pattern A (recommended): one agent wallet per process (or per subaccount).

If you can operationally afford it, the simplest way to eliminate coordination complexity is to avoid sharing a signer:

- Run one strategy instance per agent wallet, so each instance owns its nonce stream.
- If you operate multiple subaccounts in parallel, consider one agent per subaccount (or one agent per process that is dedicated to that subaccount).

This converts a distributed consensus problem into a local counter problem. It also makes incident response clearer: you can revoke a single agent if one process is compromised or misbehaving.

Pattern B: shared agent wallet with an atomic, centralized nonce service.

If you must share an agent key (e.g., for operational simplicity), then treat nonce issuance as a critical shared resource:

- Maintain an **atomic counter** in a single location (one process, or a small service with strong consistency).
- Every action request obtains a nonce from that allocator before signing.
- Ensure the allocator is robust to restarts by persisting its last issued value.

A practical allocator often “fast-forwards” to a timestamp-like value (for example, current milliseconds) to reduce collision risk after restarts. The exact nonce format requirements are exchange-specific;

the key is to remain within the permitted time window and to avoid reuse.

```
import time
import threading

class NonceAllocator:
    def __init__(self, persisted_high_watermark: int = 0):
        self._lock = threading.Lock()
        # Fast-forward to a time-based baseline to reduce collision risk after restart.
        now_ms = int(time.time() * 1000)
        self._next = max(persisted_high_watermark + 1, now_ms)

    def next_nonce(self) -> int:
        with self._lock:
            n = self._next
            self._next += 1
        return n
```

Important: retries must not reuse nonces.

A common mistake is to “retry the same request” by resending the same signed blob after a timeout. If the original request actually landed, that nonce is now burned, and the retry will be rejected as a duplicate. Even if the original did *not* land, blindly retrying without reconciliation can create dangerous state drift (e.g., you think you canceled but did not).

A robust approach is:

- On timeout, **do not automatically replay** the same signed action.
- Instead, **query and reconcile state** (open orders, order status, positions) and then decide the minimal corrective action.
- If you must retry, re-sign a fresh action with a **new nonce**, and ensure your business logic is idempotent via client-side identifiers and reconciliation.

6.3.5 Nonce collisions: how they happen in real systems

Nonce collisions are rarely caused by a simple bug like “counter increments incorrectly.” They are usually architectural:

- **Two bot instances accidentally share the same agent key.** For example, a blue/green deployment where the old version was not fully terminated.
- **Multiple subaccounts share the same agent signer.** Since nonces are tracked per signer key, not per subaccount, parallel activity collides.
- **Stateless serverless execution.** Short-lived workers that start with the same initial nonce assumption and emit actions simultaneously.
- **Restart without persistence.** The allocator forgets the last high watermark and reuses recently used nonces, which are rejected.

If you see unexplained action rejections under load, audit “who can sign with this key” before you inspect individual strategies.

6.3.6 API wallet lifecycle risks: pruning, deregistration, and address reuse

The docs highlight an operational nuance: **agent wallet pruning/deregistration can remove nonce state.** When nonce state disappears, reusing an old agent wallet address becomes risky: you may unintentionally reintroduce nonces that the exchange no longer remembers, creating replay-like surprises or at least confusing acceptance behavior.

The recommended operational stance is explicit:

- **Generate a new agent wallet rather than reusing addresses.**

For bot operators, this is analogous to key rotation with strict uniqueness: do not bring a previously retired signing identity back into service if you cannot guarantee its historical nonce usage is still recognized the way you expect.

6.3.7 Connecting nonces and agent wallets back to system architecture

Recall the earlier separation between the Info (read) and Exchange (write) paths. Nonces and agent wallets belong entirely to the write path, and that is deliberate:

- The read path should be horizontally scalable: multiple consumers can subscribe, cache, and analyze without creating coordination hazards.
- The write path should be *serialized per signer*: one key, one nonce stream, one accountable actuator that can be paused instantly.

If you adopt this discipline early, later components become simpler:

- WebSocket user streams can be used for fast execution feedback while Info queries remain a reconciliation backstop.
- Order lifecycle handling becomes a deterministic state machine instead of a collection of ad-hoc callbacks.

In the next section, we shift from write-path coordination to read-path determinism: maintaining consistent local market state from WebSocket snapshots and deltas, including reconnect and resync behavior.

6.4 WebSocket Synchronization

REST queries are well-suited for bootstrapping and auditing, but any strategy that reacts to fast microstructure needs a streaming substrate. On Hyperliquid, that substrate is WebSocket (WS): you subscribe once and then process a continuous sequence of messages. The engineering challenge is not opening a socket; it is maintaining a *correct local mirror* of exchange state despite disconnects, bursty updates, and occasional message loss.

This section builds a disciplined approach: implement your WS consumer as a **deterministic finite state machine (FSM)** that turns

an event stream (snapshots, deltas, acks, disconnects) into a local L2 state. “Deterministic” here means: for any given sequence of messages, your local book ends in exactly one well-defined state—no hidden side effects, no ambiguous intermediate assumptions.

6.4.1 The core WS contract: subscriptions, snapshots, and streaming updates

Hyperliquid WS subscriptions follow a request/ack pattern. You send a subscribe message of the form:

```
{ "method": "subscribe", "subscription": { /* stream selector */ } }
```

The server responds with an acknowledgement (a `subscriptionResponse`). For several subscriptions, that acknowledgement can include an initial **snapshot**. In Hyperliquid’s WS semantics:

- Snapshot messages are tagged `isSnapshot:true`.
- Subsequent incremental messages are tagged `isSnapshot:false`.

That single detail determines your entire synchronization algorithm. If your code treats the first update as “just another delta,” you will build a book from an undefined baseline and silently carry incorrect levels.

6.4.2 Why an FSM beats “event handlers”

A common anti-pattern is to implement WS logic as a pile of callbacks:

- *onSnapshot* → set book
- *onDelta* → patch book
- *onClose* → reconnect

This looks fine until your process hits real-world sequencing problems: disconnects mid-flight, resubscribe storms, or deltas arriving before your application is ready. An FSM forces you to encode what you are

willing to accept in each phase. If a message arrives in the wrong phase, you either buffer it or reject it; you do not apply it to a book that is not yet defined.

A robust FSM also creates a natural place to trigger **risk-off behavior**: when the state machine believes its local mirror may be stale, you stop quoting and stop placing risk-increasing orders until resynchronization completes.

6.4.3 Local L2 book building: snapshot + deltas with “zero-copy” discipline

For quants, “zero-copy” is shorthand for: *do not rebuild the entire book on each update*. Instead, hold an in-memory representation and apply incremental changes in-place.

A practical representation for an L2 book is:

- A map/dictionary keyed by price for bids and asks (to support update-by-price deltas).
- Auxiliary sorted structures or arrays to extract top-of-book quickly.

The exact data structure depends on your language and performance budget, but the invariants are the same:

1. **Snapshot overwrites everything**: it defines the authoritative initial state for that subscription session.
2. **Each delta is a patch**: update sizes at given prices; remove levels when size becomes zero; maintain ordering.
3. **Never “guess” missing information**: if an update cannot be applied cleanly (e.g., malformed, out of sequence, or inconsistent), treat this as a desync signal and resync.

This design becomes particularly valuable for strategies that compute features from the book (imbalance, slope, queue estimates). You want the feature pipeline to read from a stable in-memory structure, not from a constantly reallocated object graph.

6.4.4 A synchronization FSM for market streams

Below is a state machine that captures the minimal, safe phases for an L2 mirror. The key idea is that your code transitions intentionally between phases rather than implicitly assuming “the book is correct.”

6.4.5 State meanings and hard rules.

Implement each state with explicit acceptance criteria:

- **Snapshot:** you subscribe and wait for the first `isSnapshot:true` message. You do not publish a “ready” book until the snapshot is parsed and validated (e.g., bids/asks are sorted or can be sorted; no negative sizes; no duplicate levels unless your parser merges them).
- **Delta Updates:** only after the snapshot is committed do you apply `isSnapshot:false` updates. If you see another snapshot, treat it as authoritative (often this happens after resubscribe) and rebuild.
- **Desync Detection:** any condition that makes you unsure the mirror is correct (disconnect, malformed update, local invariant violation) moves you into an alert state. In this state, you should generally stop emitting new quotes or at least stop increasing risk.
- **Resync:** reconnect, resubscribe, and rebuild from a fresh snapshot. If you have additional read-path tools, you can also query the Info API to fill gaps, but the WS snapshot is your primary reset mechanism.

The most important discipline is psychological: treat the local book as *possibly wrong* unless your FSM says it is synchronized.

6.4.6 Disconnects are normal: engineer for periodic drops

Hyperliquid’s docs explicitly warn that disconnects can happen periodically without announcement. Your reconnect logic should therefore be part of the mainline design, not an exception handler.

A production reconnect loop typically includes:

- **Exponential backoff with jitter** to avoid thundering herds.
- **Resubscription idempotency:** your code can safely re-send subscription messages after reconnect.
- **Authoritative resync point:** after reconnect, treat the snapshot in the subscription acknowledgement (or subsequent `isSnapshot:true`) as the new base state.

Because missed data during reconnect is addressed by snapshot semantics, you should not attempt to “patch” a stale book with guesses. Reset cleanly, then resume.

6.4.7 Heartbeats and liveness

WS liveness is a two-sided contract:

- The server closes a WS connection if it hasn’t sent a message to the client in the last 60 seconds.
- Clients can send heartbeats: `{"method": "ping"}`, and the server responds with `{"channel": "pong"}`.

Even if your subscriptions are active, there can be quiet periods (e.g., an illiquid market stream or filtered subscription). A simple heartbeat loop avoids surprise disconnects and gives you a clear liveness signal for monitoring.

```
import time
PING = {"method": "ping"}
```

```
def heartbeat(ws, interval_s=20):
    last = time.time()
    while True:
        now = time.time()
        if now - last >= interval_s:
            ws.send_json(PING)
            last = now
        # Your receive loop should update liveness metrics on "pong".
        time.sleep(interval_s)
```

6.4.8 WS post: request/response correlation without leaving the socket

In addition to subscriptions, Hyperliquid WS supports **post requests** for both info queries and signed actions. The wire shape is a wrapper:

- You send a message with `method:'post'` and a numeric id.
- Responses arrive on `channel:'post'` and include the same id.

This is a latency and operational convenience: you can keep a single WS connection for streaming plus occasional point queries or action submissions. The engineering requirement is strict correlation: every outstanding id must be tracked until a response arrives or times out.

```
const msg = {
  method: "post",
  id: 123456, // unique per outstanding request
  request: {
    type: "info", // or "action" for signed actions
    payload: { /*... */ }
  }
};
ws.send(JSON.stringify(msg));
```

When you implement this, impose a cap on inflight posts and time them out deterministically; otherwise you can deadlock your own event loop under burst traffic.

6.4.9 Rate limits: design within WS constraints

Hyperliquid enforces WS limits that matter for bot architecture: caps on number of connections, number of subscriptions, messages per minute, and inflight post messages. You do not need the exact thresholds memorized to build correctly, but you do need the design implications:

- Prefer **few long-lived connections** over many short-lived ones.
- Multiplex subscriptions thoughtfully: subscribe only to what you consume.
- Batch internal computation; avoid sending chatty control messages.
- Keep write-path actions on a controlled queue so spikes do not trigger WS message-rate failures.

Rate limits are not merely a compliance issue; they shape the stability of your synchronization loop under stress.

6.4.10 Putting it together: a state sync loop you can reason about

A clean WS synchronizer can be summarized as a loop with explicit phases:

1. Connect WS; start heartbeat and liveness monitoring.
2. Subscribe; await ack and first `isSnapshot:true`; build local L2 state.
3. Transition to streaming; apply `isSnapshot:false` deltas in-place; publish a “book ready” flag for strategies.
4. On disconnect or invariant violation: move to desync state; disable risk-increasing execution; reconnect and rebuild from snapshot.

This FSM approach is intentionally conservative. It favors correctness over continuous quoting during ambiguous conditions, which is typically the right trade for on-chain perps where a single stale-book quote can produce adverse selection and unexpected inventory.

6.5 Order Lifecycle and Reconciliation

A trading system lives in two realities at once:

- **Client intent:** what your strategy *tried* to do (place, cancel, replace).
- **Exchange truth:** what the venue *actually* accepted, matched, partially matched, or rejected.

Because requests and confirmations are asynchronous—and because WebSocket connections can drop—you should assume these realities diverge occasionally. The goal of reconciliation is to close that gap quickly and deterministically, without placing duplicate orders or losing track of risk.

This section lays out a practical order state model, explains what the Exchange API returns, and provides a reconciliation playbook that uses both WS user streams and Info queries such as `openOrders` and `orderStatus`.

6.5.1 Order identifiers: exchange `oid` versus client `cloid`

Hyperliquid supports two complementary identifiers for tracking orders:

- **oid:** an exchange-assigned order id (u64). You typically learn it from the action response or from user streams.
- **cloid:** an optional client order id, a 128-bit identifier represented as a 16-byte hex string. This is the safest way to correlate “I sent

an order” with “the exchange is talking about this order,” especially across reconnects.

Treat cloid as your *idempotency handle*. If your process restarts, you can re-derive which orders were intended by inspecting persisted cloids and asking the exchange for their status.

Practical rule. Generate cloid randomly (cryptographic RNG) and persist it before submission. Never reuse a cloid for a different logical order.

6.5.2 Submission outcomes: what the Exchange API can tell you immediately

When you submit an order action, the Exchange API returns per-order status objects. At a high level, your action can result in:

- **Resting:** the order is accepted and working on the book; the response may include `{resting:{oid}}`.
- **Filled immediately:** the order matches right away; the response may include fields such as `{filled:{totalSz, avgPx, oid}}`.
- **Error / rejection:** the order is not accepted; the response includes an error string for that item.

Cancellations also return per-item success or error status.

Two cautions:

1. An “immediate” response is not the full story. Partial fills and subsequent matches arrive asynchronously via user streams.
2. Network failures can hide responses entirely. Your bot must be correct even when it never receives the initial acknowledgement.

6.5.3 Time-in-force and why it changes lifecycle dynamics

Order placement supports **ALO/IOC/GTC** semantics. From a lifecycle standpoint:

- **IOC** compresses the lifecycle: it either fills immediately (possibly partially) or cancels the remainder. Your reconciliation must expect rapid transitions and “disappearing” orders that are simply completed.
- **GTC** creates long-lived working orders, which makes reconnect correctness and open-order synchronization more important.
- **ALO** introduces additional acceptance constraints (it is designed to avoid taking liquidity). If your price would cross, the order may be rejected or not rest as expected, depending on venue rules.

The key point is not the definition (covered earlier in order types), but the operational impact: your bot should not assume that “I placed a limit order” implies it will be visible in `openOrders`. With IOC and some ALO outcomes, the order may be completed or rejected without ever becoming a durable working order.

6.5.4 A client-side order state model that reconciles cleanly

You need a local state machine that can represent uncertainty. A minimal useful state model is:

- **Created:** strategy produced an intent; cloid allocated and persisted.
- **Submitted:** signed action sent; awaiting acknowledgement (or it was lost).
- **Accepted (Resting):** exchange confirmed the order is working (has oid).
- **Partially Filled:** some quantity executed; remainder may still be working.
- **Filled:** fully executed.
- **Cancel Requested:** cancel sent; awaiting confirmation.

- **Canceled:** exchange confirms order canceled (possibly with a reason).
- **Rejected:** exchange rejected the order (with a reason).
- **Unknown / Needs Reconcile:** local uncertainty due to disconnect/timeouts.

Two principles make this state model robust:

1. **Never downgrade certainty.** Once you have confirmed Filled, do not revert to Resting because an old message arrives late.
2. **Treat “Unknown” as risk-off.** If you cannot prove the order state, do not assume it is canceled or absent; reconcile before placing offsetting orders that could double exposure.

6.5.5 User streams: the fast path for reconciliation

WS user streams provide near-real-time feedback for what happened to your orders. The key streams mentioned in the docs include:

- `orderUpdates`
- `userEvents`
- `userFills`

A robust execution engine typically uses:

- `orderUpdates` to move orders among Accepted, Canceled, Rejected, and sometimes fill-related states.
- `userFills` as the authoritative execution ledger: fills change position and PnL; they are what you ultimately care about for risk.

Because WS can disconnect, user streams are the *fast path*, not the only path. Your design should tolerate missing an update and repairing later via Info.

6.5.6 Info queries: the authoritative backstop

Two Info requests are particularly useful as reconciliation primitives:

- `openOrders`: returns the current working set of open orders for the user address.
- `orderStatus`: fetches granular status for a specific order, queried by oid (u64) or cloid (16-byte hex string).

`orderStatus` returns detailed states including open, filled, canceled, triggered, and rejected, and it can also surface specialized cancel/reject reasons such as:

- `selfTradeCanceled`
- `reduceOnlyCanceled`
- `openInterestCapCanceled`
- `scheduledCancel`
- `tickRejected`
- `minTradeNtlRejected`

These reasons are not just diagnostics; they are inputs into adaptive behavior (round price to tick, adjust size to meet minimum notional, coordinate strategy instances to avoid self-trade cancellation, etc.).

6.5.7 A reconciliation playbook (battle-tested workflow)

The following playbook is a practical way to keep client intent and exchange truth aligned:

Step 0: persist intent before sending. When you decide to place an order:

- generate cloid;

- write an “intent record” to durable storage (even a local WAL);
- then sign and submit.

If the process dies after submission but before persisting, you cannot later prove whether the order existed.

Step 1: capture immediate action response (best-effort). On receiving the action response:

- if resting appears, store oid and mark Accepted (Resting);
- if filled appears, store oid, record the fill summary, and mark Filled (subject to later fill ledger confirmation);
- if an error appears, mark Rejected and store the error string.

Step 2: listen continuously to WS user streams. As events arrive:

- update local order state;
- record fills from `userFills` to your execution ledger;
- update derived risk (net position, exposure) from the fill ledger, not from assumptions.

Step 3: periodically reconcile with Info. On a timer (and immediately after any WS reconnect), do:

1. Call `openOrders` for the user address and reconcile the working set:
 - Any locally Accepted order missing from `openOrders` becomes Unknown / Needs Reconcile.
 - Any order in `openOrders` that the client does not recognize indicates either a bookkeeping bug or a manual intervention; alert and quarantine execution.
2. For each Unknown or “stuck” order, call `orderStatus` by cloid (preferred) or oid:

- transition to Filled/Canceled/Rejected accordingly;
- store the cancel/reject reason for debugging and strategy adjustment.

Step 4: correct state with minimal actions. Only after you have a consistent picture should you issue corrective cancels or replacement orders. This prevents the classic failure: a bot assumes an order is canceled, places a replacement, and later discovers both were live.

6.5.8 Troubleshooting matrix: “why did my order disappear?”

When an order you expected to be working is not present in openOrders, do not guess. Use orderStatus and interpret the reason:

Observed	Likely cause / response
tickRejected	Your price does not respect tick size. Apply deterministic tick rounding (and ensure rounding direction is consistent with isBuy).
minTradeNtlRejected	Size $\times$ price is below minimum notional. Enforce min notional checks before signing.
selfTradeCanceled	Two strategies (or two processes) crossed against each other. Coordinate quoting across instances; consider a single execution gate per symbol.
reduceOnlyCanceled	Order would increase position when reduceOnly is set. Fix the reduce-only policy logic: compute current position and clamp size.
openInterestCapCanceled	Venue-wide or asset-level cap constraint hit. Add a fallback behavior: reduce size or skip until conditions change.
scheduledCancel	Venue performed a scheduled cancellation (operational event). Immediately resync, then re-evaluate whether to re-place orders.

This matrix is also a testing checklist: you can intentionally trigger many of these paths on testnet to verify that your bot records reasons and reacts safely.

6.5.9 Idempotency: what you can and cannot guarantee

Because Exchange actions are nonce-based and state-changing, “retry” is never a purely transport-layer decision. Idempotency must be de-

signed at the application layer:

- cloid **gives you semantic idempotency for order intent.** If you lose an acknowledgement, you can query by cloid to learn what happened without duplicating the order.
- **Nonces prevent replay but do not solve duplicate intent.** Two different signed actions with two different nonces can still place two orders if your strategy emits intent twice.

A simple but powerful discipline is: *exactly one code path is allowed to originate intents*, and it must persist them. Everything else (retries, reconnect repairs) is a reconciliation loop that converges local state to exchange truth rather than blindly resubmitting.

6.5.10 Putting reconciliation into your execution engine

A production execution engine typically has three cooperating loops:

- **Intent loop:** strategy emits desired target orders (prices/sizes), tagged with fresh cloids.
- **Execution loop:** sends signed actions, enforces rate limits and nonce allocation, and records immediate responses.
- **Reconciliation loop:** consumes WS user streams continuously and periodically calls `openOrders/orderStatus` to repair gaps.

This is the point where earlier sections pay off: signed actions are safe (cryptographic authentication), concurrent writes are coordinated (nonces and agent wallets), and your read path can recover after disconnects (WS synchronization). With these foundations, the remaining design choice is mostly about implementation language and SDK stack, which we address next.

6.6 Language Selection and SDK Ecosystem

By this point, the technical requirements of a Hyperliquid trading system are concrete: you need a read path that can stay synchronized over WebSockets, and a write path that signs actions correctly, allocates nonces safely under concurrency, and reconciles order state over time. Language choice and SDK selection are not stylistic preferences; they directly affect whether you can meet those requirements without introducing subtle failure modes.

Hyperliquid documentation explicitly points to an **official Python SDK** and lists **community Rust/TypeScript SDKs** plus **CCXT integrations**. The ecosystem gives you two broad paths:

- **Native (Hyperliquid-aware) SDKs:** expose venue-specific primitives such as signing schemes, agent wallet handling, nonce conventions, and WS subscription idioms.
- **Generic (cross-exchange) abstractions such as CCXT:** provide portability across venues (`createOrder`, `cancelOrder`, `fetchMarkets`, ...), but may hide or omit Hyperliquid-specific behavior that matters for correctness and performance.

This section compares Python, TypeScript, and Rust through the lens of latency, type safety, and concurrency, then closes with practical guidance on when CCXT is an advantage versus a liability.

6.6.1 Start from requirements, not from language preference

A useful way to select a stack is to map each earlier engineering challenge to a capability:

- **Signing correctness (high risk):** the signing scheme must match exactly (e.g., `sign_l1_action` vs `sign_user_signed_action`). Any mismatch causes hard-to-debug rejections.

- **Nonce allocation (coordination):** one signer key implies one nonce stream; parallelism must be controlled.
- **WS synchronization (stateful streaming):** you need a deterministic consumer that can resubscribe, process `isSnapshot:true` then `isSnapshot:false`, and recover from disconnects.
- **Order reconciliation (asynchrony):** you must integrate user streams with Info backstops and maintain a local state machine.

Language and SDK choices mainly change how hard each requirement is to implement *correctly* and how expensive it is to operate under load.

6.6.2 Python: fastest path to correctness and iteration

For most traders, the official Python SDK is the shortest route to a production-grade integration because it provides turnkey signing and example scaffolding, and it encodes operational conventions (environment constants and API wallet handling). That “convention encoding” matters: it reduces the number of decisions you have to get exactly right.

Where Python shines.

- **Signing reliability:** using the official SDK reduces the probability of byte-level serialization mistakes.
- **Development speed:** rapid iteration for strategies, research loops, and testnet experimentation.
- **Operational clarity:** Python’s readability helps when you are debugging reconcile edge cases at 3 a.m.

Where Python becomes limiting.

- **Latency-sensitive market data loops:** pure Python can struggle with very high message rates unless you are careful about allocations and data structures.

- **Concurrency discipline:** Python makes it easy to accidentally run multiple event loops or worker processes that share a signer key, which can create nonce collisions unless you impose strict architecture.

A common hybrid pattern is to use Python for orchestration and strategy logic while pushing the tightest data-path code (book maintenance, feature extraction) into optimized components or a separate service.

6.6.3 TypeScript: strong fit for Node services and front-end integration

TypeScript (and JavaScript runtimes) often appear in two places:

- **Node-based execution services:** event-driven WS consumers, lightweight execution gateways, and internal APIs.
- **Front-end / dashboard tooling:** monitoring, operator controls, and user-facing analytics.

TypeScript's advantage is that it naturally fits WS-heavy, event-driven architectures, and its type system can improve maintainability compared to plain JavaScript.

Trade-offs.

- **Type safety is only as good as your schemas.** If the SDK types are incomplete, you can still mis-handle messages (e.g., treating snapshots and deltas interchangeably).
- **Performance is good enough for many strategies, but not all.** High-frequency book processing can be viable, but you must be careful with garbage creation and object churn in tight loops.

TypeScript is often a good choice when you want one codebase to power both execution services and operator interfaces, and when you accept that the deepest low-latency optimization may require additional work.

6.6.4 Rust: performance, memory safety, and concurrency control

Rust is the natural choice when your bottleneck is *throughput* and *tail latency*, especially for:

- maintaining local L2 mirrors with minimal allocations,
- computing microstructure features at high frequency,
- running many symbols in parallel with strict control over shared resources (like nonce allocators).

Rust's strong type system and ownership model reduce entire classes of runtime errors (use-after-free, data races), which matters when your bot is long-lived and stateful.

Rust costs.

- **Developer time:** implementing and integrating all protocol details is more time-consuming unless the SDK is mature.
- **Protocol correctness risk:** if you do not rely on a well-tested signing implementation, you can reintroduce the signature mismatch problems that SDKs are meant to prevent.

In other words, Rust improves *engineering guarantees* once the integration is correct, but it does not automatically reduce the work of achieving correctness in the first place.

6.6.5 Native SDKs versus CCXT: portability versus venue-specific control

CCXT is valuable when you care about **portability** across exchanges and want a unified interface (`createOrder`, `cancelOrder`, `fetchMarkets`, etc.). For a multi-venue research stack, this can be a real productivity boost.

However, CCXT's abstraction can become a disadvantage on Hyperliquid because Hyperliquid's operational details matter:

- agent wallets (API wallets) and how they are approved/used,
- nonce behavior and the per-signer coordination it imposes,
- specialized order status taxonomy and granular cancel/reject reasons,
- certain WS user streams that are essential for fast reconciliation.

A generic wrapper may either (i) not expose these features, (ii) expose them awkwardly, or (iii) lag the exchange’s feature set. If your strategy depends on those details—for example, using `clound-centric idempotency` or reacting differently to `tickRejected` versus `selfTradeCanceled`—a native SDK tends to be the safer baseline.

A pragmatic rule.

- Use **native SDKs** when you are building a production bot that relies on venue-specific behavior for safety and performance.
- Use **CCXT** when your goal is cross-venue prototyping, portfolio-level analytics, or when execution logic is deliberately simple and you can tolerate abstraction loss.

6.6.6 Decision table: choosing a stack for different bot profiles

Bot profile	Recommended stack and rationale
Research / discretionary automation	Python + official SDK for fastest iteration and lowest signing risk; add WS sync and reconciliation loops as described earlier.
Single-venue systematic execution	Python or TypeScript with native SDK; prioritize correctness (signing, nonces, reconciliation) over micro-optimizations.
Latency-sensitive quoting across many symbols	Rust (or Rust service) for WS processing and feature computation; still prefer SDK-backed signing logic or thoroughly tested signing modules.
Multi-venue strategy prototypes	CCXT for portability, with the understanding that Hyperliquid-specific features may need direct API calls alongside CCXT for full control.

6.6.7 A minimal “hello world” checklist for any language

Regardless of the language, you should validate the same three capabilities on testnet before touching mainnet:

- **One Info query:** fetch a small piece of state and parse it reliably.
- **One WS subscription:** handle `subscribe` → `snapshot` (`isSnapshot:true`) → `deltas` (`isSnapshot:false`) and survive forced disconnects.
- **One signed action:** place a tiny order with a persisted cloid, confirm acknowledgement, and reconcile final state using user streams plus `orderStatus`.

If your chosen SDK or wrapper makes any one of these steps opaque, difficult to test, or hard to debug, that is an early warning sign. For on-chain perps, production stability is usually earned through explicit control over signing, nonces, and reconciliation—not through hidden abstractions.

6.6.8 Closing perspective

Language choice is ultimately about *where you want to spend complexity*:

- Python spends complexity on performance tuning, but saves it on correctness and speed to deployment.
- Rust spends complexity on development effort, but buys you deterministic performance and strong concurrency guarantees.
- TypeScript sits between, with strong event-driven ergonomics and good maintainability when types and schemas are treated as first-class assets.
- CCXT buys portability, but you should assume you will still need direct Hyperliquid-native access for full operational control.

The right choice is the one that lets you implement the earlier sections' safety properties—signing correctness, nonce discipline, WS synchronization, and reconciliation—with the fewest moving parts in your specific operating environment.

Chapter 7

Bot Architecture, Operations, and Production Security

A winning quantitative strategy is only as good as the infrastructure executing it. In the high-stakes environment of on-chain perpetuals, a microsecond of latency or an unhandled WebSocket disconnect can evaporate weeks of alpha. This chapter bridges the critical gap between theoretical models and battlefield-tested trading infrastructure. You will learn how to architect decoupled event-driven systems, forge resilient error-handling frameworks to survive network volatility, and lock down your operational security. We are moving beyond building bots that simply trade to engineering relentless machines built to survive, scale, and rigorously protect your capital in production.

7.1 Event-Driven Bot Architecture

A production bot on Hyperliquid is not a single loop that “reads prices and places orders.” It is a small distributed system that happens to run

in one process. Market data arrives asynchronously over WebSockets; order actions go out over HTTPS; user streams can deliver snapshots followed by incremental updates; and network timing is adversarial precisely when the market is most volatile. If you let these concerns mix freely, you get a fragile design where a slow API call can block risk logic, where a reconnect can corrupt state, and where a strategy bug can bypass safety checks.

The architectural goal is *decoupling*: isolate (i) **alpha generation** (what you want to trade), (ii) **risk rules** (what you are allowed to do), and (iii) **I/O execution** (how you talk to the exchange). Decoupling is not stylistic; it is a control mechanism. It ensures that failures, latency spikes, and partial outages in one layer do not automatically become trading losses.

7.1.1 The core principle: make time explicit with events

A synchronous design pretends time is linear: “read market, compute, send order, wait.” On-chain perps do not cooperate with that narrative. You must assume:

- data arrives out of order (due to reconnects, snapshots, or buffering),
- actions acknowledge late (HTTP timeouts can hide success),
- the “truth” is sometimes only recoverable by polling `/info`.

Event-driven architecture makes time explicit. Everything important becomes an **event** that is published onto a bus and handled by subscribers. Handlers can be retried, throttled, or temporarily paused without changing upstream producers. This is how you prevent slow I/O from blocking alpha generation or risk checks.

A useful mental model is: *strategies do not trade; they propose. Risk does not execute; it authorizes. Execution does not decide; it routes.*

7.1.2 Define contracts: events, state, and intents

Decoupling works only when boundaries are crisp. In practice you want three kinds of messages:

(1) External events (facts).

These are statements about the world, sourced from WebSocket feeds or polling endpoints:

- market microstructure updates (trades, top-of-book, depth),
- user events (fills, funding),
- rate limit and account/margin context (from /info calls such as `userRateLimit` when you choose to poll it).

These events should be treated as *immutable*; your system may later discover they were incomplete (missed during disconnect), but it should not retroactively rewrite them.

(2) Internal state updates (derived).

These are the bot's own computed views:

- consolidated order state (open/filled/canceled) as best-known,
- position, PnL, and inventory (as discussed earlier),
- feature vectors used by alpha (e.g., spreads, imbalance).

A key rule: **state is owned by a dedicated state component**, not by the strategy. Strategies can read state (or subscribe to derived events), but they should not be the authoritative ledger.

(3) Trading intents (requests).

These are proposed actions that are not yet orders:

- `PlaceIntent`: “post a buy at price p size q with time-in-force X ”,
- `CancelIntent`: “cancel order with CLOID c ”,
- `ReplaceIntent`: “modify from (p_0, q_0) to (p_1, q_1) ” (implemented as cancel/replace depending on exchange semantics).

Intents are produced by alpha and consumed by risk. Risk converts them into *authorized order commands* or rejects/clips them.

7.1.3 A practical event taxonomy

Category	Examples	Notes for decoupling
Market events	trade, book update	Never block on them; ingest fast, process downstream.
User events	fills, funding, order updates	Hyperliquid user streams include an initial snapshot (<code>isSnapshot:true</code>) then incremental messages (<code>isSnapshot:false</code>); treat snapshot as a state reset input.
Control events	reconnect, heartbeat, lag	Promote to first-class events so strategy/risk can degrade safely.
Risk context	margin/position refresh	Often best as periodic polls or reconciliation triggers; keep separate from alpha logic.
Intents	place/cancel/replace	Not exchange-specific; an intent is a proposal, not a side effect.
Commands	signed order actions	Exchange-specific, rate-limited, retried; only the execution engine emits these.

7.1.4 The message bus: buffer, sequence, and apply backpressure

The message bus is the glue. It can be implemented as:

- in-process queues (common for single-binary bots),
- a lightweight pub/sub abstraction (topics),
- or an external broker (rarely needed unless you split services).

Regardless of implementation, the bus should provide three operational properties:

Buffering. During transient spikes (e.g., volatility bursts), you want queues to absorb load. The alternative is dropping events or blocking producers, both of which create unpredictable strategy behavior.

Sequencing. Within a topic, you want a monotonic ordering (often “arrival order”), plus metadata such as timestamps and sequence numbers when available. Your handlers should be able to detect gaps or regression (e.g., a snapshot arrives after increments).

Backpressure. If consumers cannot keep up, the bus must surface that fact. Backpressure signals are themselves useful events: if your book handler lags by 2 seconds, the correct trading response may be to stop quoting rather than to continue with stale inputs.

A simple backpressure rule that keeps systems sane:

If `queue_depth` or `event_lag` exceeds a threshold, reduce activity rather than “speed up” with more threads.

More concurrency can increase contention and tail latency; it does not magically remove upstream API limits or network delays.

7.1.5 Ingestion: normalize early, decide late

Ingestion is where raw Hyperliquid messages are converted into internal events. The discipline here is: **do minimal work in the network callback**. Parse, validate, timestamp, and publish.

Two Hyperliquid-specific ingestion concerns deserve explicit design:

7.1.6 Snapshot-first streams: hydrate, then increment

User streaming endpoints send an initial snapshot with `isSnapshot:true` and then incremental updates with `isSnapshot:false`. Treat this as a state machine in your ingestion/state layer:

- On snapshot: replace the relevant portion of state (orders, fills, positions) in one atomic update.
- After snapshot: apply incrementals in order, rejecting incrementals that precede hydration.
- On reconnect: assume you missed incrementals and require a new snapshot to rehydrate.

If you let strategies read user state before hydration, you invite a class of bugs where the strategy trades on “empty” positions, then over-levers, then risk scrambles to compensate.

7.1.7 Normalize identifiers for idempotency (OID vs CLOID)

Hyperliquid supports `/info orderStatus` queries by *oid* (a u64) or by *cloid* (a 16-byte hex client order id). Architecturally, this implies:

- **execution owns CLOIDs:** it creates them, persists them, and uses them for idempotent retries;
- **state records both:** when an acknowledgment or status reveals the oid, bind it to the original cloid;
- **reconciliation prefers CLOID:** after disconnects, CLOID lets you ask “did my intent become a live order?” without relying on having received the oid in real time.

7.1.8 State ownership: one ledger, many readers

A common failure mode in bot code is “state by convenience”: each module keeps its own partial copy of positions or open orders. In calm markets, these copies look consistent; under reconnections, cancellations, and partial fills, they diverge.

Instead, designate a single **state store** component that:

- ingests external events and produces derived state updates,
- answers read-only queries for strategy and risk (synchronously in-memory),
- persists checkpoints so the bot can restart safely.

7.1.9 Persistence: checkpoints, not journals (unless you need replay)

You have two workable persistence approaches:

Checkpointing (simpler). Periodically write the minimal sufficient state to disk:

- latest known positions and balances,
- open orders (CLOID/OID mapping, price/size, timestamps),
- last processed sequence markers per feed (if available),
- configuration/version hash (so you can detect incompatible restarts).

On restart, you load the checkpoint, then immediately reconcile with /info and snapshots.

Event journaling (more powerful). Append all events to a log and replay to rebuild state. This is valuable for research and debugging, but it increases engineering load (log growth, schema evolution, replay determinism). Many production bots use checkpointing in production and journaling in simulation.

The operational invariant is the same: **after a restart, do not trade until the state store is rehydrated and reconciled.**

7.1.10 Alpha engine: produce intents, never touch the wire

Alpha generation should be a pure transformation:

(state, market events, parameters) → intents.

Purity here does not mean “no latency.” It means *no side effects*. The alpha engine should not:

- call /exchange,
- manage retries,
- handle reconnects,
- update authoritative positions.

This separation buys you three things:

- You can unit test alpha with recorded event streams (deterministically).
- You can run alpha in a tight loop without being blocked by I/O.
- You can apply the same risk policy to multiple alphas consistently.

A practical design pattern is to make alpha emit *desired state* rather than individual orders. For example: “target inventory I^* ” or “target quote ladder around mid.” A separate quoting module translates that into specific `PlaceIntent/CancelIntent` messages, which are then risk-checked.

7.1.11 Risk engine: a gate with memory

Risk is not a one-line “max position” check. It is a *stateful policy layer* that:

- reads current exposure and open-order risk from the state store,
- validates each intent against limits (inventory, notional, leverage, concentration),
- can clip size, widen prices, or reject entirely,
- enforces global modes (e.g., “reduce-only” after anomalies),
- emits only authorized order commands to execution.

Two implementation details matter for decoupling:

Risk should be fast and deterministic. If risk is slow, it becomes the bottleneck that forces you to either (i) skip checks, or (ii) block alpha. Keep risk computations in-memory and avoid network calls. If risk needs external truth (e.g., margin state), treat that truth as an event from ingestion/reconciliation, not a synchronous dependency.

Risk should be the only place that converts intents into exchange-specific commands. This prevents “strategy bypass,”

where a new strategy accidentally sends orders directly. In code review and auditing, a single choke point is priceless.

7.1.12 Execution engine: asynchronous I/O with idempotent order management

Execution is where decoupling meets reality: network calls can fail, time out, or be rate limited. The execution engine must therefore be:

- **asynchronous:** it should accept commands and return immediately (ack later via events),
- **idempotent:** the same command retried twice should not create two unintended orders,
- **observable:** every request produces structured logs/metrics and correlates to CLOUDs.

A minimal execution loop uses a work queue of authorized commands. Each command has:

- a unique cloid (generated by execution),
- a local deadline (timeout),
- a retry budget (bounded),
- an `expiresAfter` value that must be kept correct (because stale `expiresAfter` can cause cancels and consume a higher rate-limit cost when rejected).

7.1.13 Single point of retry

Retries belong in one layer only: execution. If strategy retries and execution retries, you get multiplicative retry storms. A clean contract is:

- upstream publishes one command,
- execution either confirms (via user stream events / reconciliation) or emits a failure event after exhausting policy.

7.1.14 Reconciliation: repair truth after partial failure

Even with perfect WebSocket handling, you must plan for:

- disconnects during fills,
- acks that arrive after your local timeout,
- missed user events.

Reconciliation is a background loop that polls `/info` to answer: *what is the exchange's current view of my orders?* Hyperliquid provides order-Status queries by oid or cloid, which enables a robust repair workflow:

1. Identify “uncertain” orders (commands sent but not confirmed).
2. Query `/info orderStatus` for those CLOUDs.
3. Update the state store to the returned status.
4. If the order is open but violates current risk (e.g., risk mode changed), issue a cancel command.

The critical discipline is: **reconciliation updates state; it does not invent fills**. All PnL and position changes should flow from user events or authoritative `/info`-backed state transitions.

7.1.15 A concrete skeleton: queues between components

Below is a simplified (and deliberately incomplete) skeleton illustrating how decoupling looks in code organization. The point is not the language runtime but the direction of dependencies: strategy depends on state, not on HTTP; execution depends on commands, not on strategy internals.

```
from dataclasses import dataclass
from typing import Optional, Any

# --- Events (facts) ---
```



```
@dataclass(frozen=True)
class MarketEvent:
    ts_ns: int
    payload: Any

@dataclass(frozen=True)
class UserEvent:
    ts_ns: int
    isSnapshot: bool
    payload: Any

# --- Intents (proposals) ---
@dataclass(frozen=True)
class PlaceIntent:
    symbol: str
    side: str
    price: float
    size: float

@dataclass(frozen=True)
class CancelIntent:
    cloid: str

# --- Commands (authorized, exchange-specific) ---
@dataclass(frozen=True)
class OrderCommand:
    cloid: str
    kind: str # "place" / "cancel" / "replace"
    body: dict
    expiresAfter: int

class StateStore:
    def apply_event(self, ev: MarketEvent | UserEvent) -> None:...
    def snapshot_ready(self) -> bool:...
    def view(self) -> dict:...

class AlphaEngine:
    def on_market(self, state_view: dict, ev: MarketEvent) -> list[PlaceIntent |
        CancelIntent]:...

class RiskEngine:
    def authorize(self, state_view: dict, intent: PlaceIntent | CancelIntent) -> Optional[
        OrderCommand]:...

class ExecutionEngine:
    async def submit(self, cmd: OrderCommand) -> None:...
```

This structure makes two operational behaviors straightforward:

- You can halt execution (e.g., due to rate limits or incidents) while still ingesting and updating state.
- You can disable alpha while keeping risk/execution alive long enough to cancel stale orders.

7.1.16 Graceful recovery: startup, reconnect, and shutdown sequences

Decoupled systems still need *coordinated* lifecycle behavior. Codify these sequences as explicit states in your bot runtime (even if you do not formalize them as a state machine).

7.1.17 Startup (cold start)

A robust startup sequence is:

1. Start ingestion; connect WebSockets.
2. Wait for user stream snapshot(s) with `isSnapshot:true`.
3. Initialize state store from snapshots (and/or last checkpoint).
4. Run a reconciliation pass with `/info orderStatus` for any persisted open CLOIDs (or query relevant order sets, depending on your stored data).
5. Only then enable alpha intent publication.

This prevents “blind trading” where the bot places new orders before it knows about existing exposure.

7.1.18 Reconnect (hot recovery)

On WebSocket reconnect:

- immediately mark state as degraded and block new intents at the risk gate,
- demand fresh snapshots for user streams,
- run reconciliation for uncertain orders,
- re-enable trading only after state is coherent again.

This is not paranoia; it is acknowledging that any missed fill can flip your inventory sign, turning “spread capture” into a directional bet.

7.1.19 Shutdown (safe stop)

A safe shutdown should:

- stop alpha publication first,
- let execution drain or cancel pending commands,
- persist a final checkpoint (open CLOIDs/OIDs, last-seen timestamps),
- close sockets last.

The ordering matters: shutting down ingestion first removes your ability to observe fills/cancels that happen during the shutdown window.

7.1.20 What decoupling buys you in practice

When the components are truly separated by an event bus and strict contracts, you gain:

- **Deterministic debugging:** you can replay recorded events into state/alpha/risk without touching the network.
- **Bounded blast radius:** if /exchange becomes slow, execution queues grow and risk can tighten, but state and monitoring continue.
- **Operational safety:** reconciliation can repair missed events using /info orderStatus by oid or cloid, instead of guessing.
- **Composable strategy research:** you can run multiple alpha modules that emit intents into the same risk/execution pipeline, making comparisons fair and reducing duplicated code.

Most importantly, the architecture enforces a hierarchy of responsibility: alpha proposes, risk constrains, execution acts, and state records. In volatile on-chain perps, that separation is the difference between a temporary network issue and an unrecoverable loss.

7.2 Rate Limits and Error Resilience

An on-chain perp bot fails in ways that discretionary traders rarely experience: not just “bad fills,” but partial outages, reconnect storms, request throttling, and ambiguous order outcomes (you timed out locally, yet the exchange may have accepted the order). The operational goal in this section is narrow and concrete: build a *deterministic* framework for timeouts, retries, and circuit breakers so your bot behaves predictably under stress instead of amplifying the stress.

The architectural decoupling from the prior section is what makes this feasible. With ingestion, state, risk, and execution separated, resilience mechanisms can be placed precisely where they belong: **around external dependencies** (HTTP/WebSocket I/O), not scattered through alpha and risk logic. This avoids the classic mistake of “retry everywhere,” which turns a transient incident into a self-inflicted outage.

7.2.1 Start with a failure taxonomy: transient, terminal, and ambiguous

Every error-handling policy implicitly answers two questions:

- *Should I try again?*
- *If I try again, will I accidentally do the action twice?*

To make those answers consistent, classify outcomes into three buckets.

Transient failures (retryable). These are failures where the request did not succeed and repeating later has a reasonable chance of working:

- network flakiness (disconnects, DNS hiccups),
- server-side overload (often surfaced as 5xx),
- rate limiting such as HTTP 429 (retryable, but only with backoff and throttling).

Terminal failures (non-retryable without a code/config change). These are failures where repeating the request as-is is unlikely to succeed and may be harmful:

- invalid parameters (e.g., malformed order fields),
- authorization failures (wrong credentials / signing problems),
- policy rejections that will repeat until risk/logic changes.

The right response is to emit a high-signal error event, stop placing new orders for that route, and let the operator or a higher-level safety controller decide next steps.

Ambiguous outcomes (“unknown success”). This is the most important category for trading: your client did not receive a definitive response (timeout, broken connection), so you cannot tell whether the action executed.

In ambiguous cases, **do not blindly retry order submission**. Treat it as “unknown” and reconcile using `/info orderStatus` by oid or cloid. This turns a potentially catastrophic double-submit into a controlled state-repair operation.

7.2.2 One retry layer, bounded budgets, and short timeouts

The core systems principle is:

Retry at a *single* point in the stack, with explicit budgets; otherwise, retries multiply and outages worsen.

In the bot architecture from the previous section, the natural location for this single retry layer is the **execution engine**. Alpha and risk should never retry network calls; they should only react to events such as “command failed” or “command status unknown.”

7.2.3 Timeouts: fail fast so you can recover

A timeout is not pessimism; it is resource management. Without timeouts, hung sockets or stalled HTTP calls pin your event loop, build up

queues, and create stale decisions. With timeouts, you can:

- mark the command as ambiguous,
- start a reconciliation query,
- temporarily reduce trading intensity (or halt new commands) if ambiguity accumulates.

Practical guidance:

- Use **short** timeouts for order placement/cancellation (your bot cares about timely feedback).
- Use **separate** timeouts for informational polling (e.g., /info) so background tasks cannot starve order management.

7.2.4 Retry budgets: cap both count and time

A resilient bot caps retries in two dimensions:

- **attempt budget**: max attempts per command (e.g., 3 tries),
- **time budget**: max wall-clock time spent retrying before giving up (e.g., 2 seconds).

These budgets prevent a single stuck dependency from consuming the entire bot. They also create a stable worst-case behavior that you can reason about during fast markets.

7.2.5 Backoff with jitter: avoid synchronized retry bursts

Exponential backoff is necessary but not sufficient. If many bots (or many tasks inside your bot) fail at the same time and all retry on the same exponential schedule, they can re-synchronize and create periodic load spikes. This is the “thundering herd” problem.

The standard fix is **jitter**: add randomness to spread retries over time. The specific strategy recommended in modern distributed-systems practice is *Full Jitter*:

$$\text{sleep} \sim \text{Uniform}(0, \min(\text{cap}, \text{base} \cdot 2^k))$$

where k is the retry attempt number.

Why this works:

- exponential growth reduces pressure on a struggling service,
- randomness prevents alignment across clients and across your own concurrent tasks.

7.2.6 Determinism where it matters

“Jitter” sounds non-deterministic, but you can still make your system operationally predictable:

- keep the *policy* deterministic (same inputs $\rightarrow$ same retry budget and cap),
- log the chosen sleep durations and random seed per process instance,
- keep randomness confined to timing, not to trading decisions.

7.2.7 Adaptive throttling using `/info userRateLimit`

Static rate limits (“I will send at most N requests per second”) are rarely optimal because real capacity can vary with market conditions and platform load. Hyperliquid exposes `userRateLimit` via `/info`, returning request usage and capacity fields; bots can poll this to adaptively throttle rather than relying on static limits.

This enables a control loop:

- Poll `/info userRateLimit` on a schedule (or when 429s occur).
- Compute a utilization ratio (used vs cap).

- Adjust your local rate limiter (token bucket / leaky bucket) so utilization stays below a chosen safety threshold.

Two operational details make this effective:

Treat throttling as a first-class signal. Emit an internal event such as `RateLimitPressure(high)` so strategy/risk can adapt (e.g., widen quotes, reduce churn, consolidate cancels).

Prefer reducing *order churn* over reducing *data freshness*. If you must cut request volume, cut expensive, low-value calls first (e.g., repeated cancel/replace loops driven by tiny mid-price changes). Keep the minimum set of calls needed to manage exposure safely.

7.2.8 The `expiresAfter` trap: avoid self-amplifying throttling

Hyperliquid has an important operational quirk: certain exchange actions may consume 5x the usual address-based rate limit when canceled due to a stale `expiresAfter` field. Incorrect expiry management can therefore create self-amplifying throttling: you send a command, it is rejected/canceled due to staleness, and you pay a higher rate-limit cost, which increases throttling, which causes more timeouts, which causes more stale expiries.

The design response is straightforward:

- Centralize `expiresAfter` computation inside execution (one implementation, one clock source).
- Ensure **fresh expiry on retries**: do not resend the same stale `expiresAfter`.
- Monitor and alert on any signature of stale-expiry behavior (even just a counter in logs), because it is an anti-pattern that can silently dominate your request budget.

This is a good example of why resilience belongs in execution: expiry hygiene is I/O-adjacent and should never be duplicated in strategy modules.

7.2.9 Circuit breakers: stop digging when the dependency is failing

Retries help when failures are random and brief. They harm when a dependency is persistently failing: you waste capacity, worsen congestion, and fall behind on critical tasks.

A **circuit breaker** is a policy gate that temporarily stops calls to a failing dependency. Conceptually it has three modes:

- **Closed:** calls flow normally.
- **Open:** calls are blocked immediately (fast-fail) for a cool-down period.
- **Half-open:** allow a small number of probe calls to test recovery.

You do not need a heavyweight framework to apply the pattern; what you need is consistent thresholds and clear behavior.

7.2.10 What should trip the breaker?

For a trading bot, the most useful breaker inputs are *symptoms you can measure locally*:

- consecutive failures to `/exchange`,
- sustained elevated 429 rate (rate-limit pressure that does not subside after throttling),
- timeouts exceeding a threshold within a time window,
- inability to reconcile order status via `/info`.

Be careful with overly sensitive breakers. In fast markets you will always see some errors; the breaker should trip on *clusters* that indicate systemic trouble.

7.2.11 What should the breaker do?

When the breaker is open for order actions:

- stop sending new place/replace commands (prevent churn),
- allow a limited set of *protective* actions if they are still possible (often cancels; in severe cases you may also block cancels to avoid wasting quota),
- escalate by emitting an internal high-severity event so a higher-level safety layer can reduce exposure or halt trading.

A key point: circuit breakers are not only for protecting the exchange; they protect *your bot's internal state*. If you cannot reliably send actions, your local view of open orders becomes stale. It is often better to stop and reconcile than to continue trading in a fog.

7.2.12 Bulkheads and time limiters: isolate damage within your own bot

Resilience patterns are composable: circuit breakers, bulkheads, rate limiters, retries, and time limiters can be layered (often as wrappers/decorators) to isolate failures in external dependencies.

Two patterns are especially helpful in an event-driven trading system:

Bulkheads (concurrency caps). A bulkhead limits how much concurrency a dependency can consume. For example, you might allow:

- at most 1–2 concurrent /exchange calls (to avoid request pileups),
- at most 1 concurrent reconciliation worker (to avoid /info storms).

When the bulkhead is full, new work fails fast or is deferred. This prevents a backlog of thousands of pending requests that will be irrelevant when they finally execute.

Time limiters. Even if you set HTTP timeouts, you also want higher-level deadlines: “If this command has not resolved (success, terminal

failure, or reconciled ambiguity) within T , stop trying and change trading mode.” This keeps the bot from accumulating unresolved uncertainty.

7.2.13 Idempotency and reconciliation: the safe alternative to blind retries

Trading actions are side effects. If you retry a side effect, you must control duplication.

In Hyperliquid, the practical toolkit is:

- use a unique cloid for each logical order intent,
- if order placement times out, treat it as ambiguous,
- query `/info orderStatus` by cloid to decide whether the order exists, is open, or was rejected.

This approach changes the character of error handling. Instead of “try again and hope,” you get a state-based workflow:

- **Submit** command (with timeout).
- If confirmed quickly: update state normally.
- If ambiguous: **freeze** further churn for that symbol (optional but prudent), then **reconcile** via `orderStatus`.
- If the order exists: continue from the reconciled state (possibly cancel/replace if it violates current risk).
- If it does not exist: only then consider resubmission, with a fresh `expiresAfter`.

The cost is extra code. The benefit is eliminating the most dangerous class of operational bug: accidental duplicate orders created during network turbulence.

7.2.14 A minimal, deterministic resilience policy (recipe)

To make these ideas implementable, it helps to write the policy as a small set of rules that every network call follows.

Situation	Bot behavior (execution layer)
HTTP 429	Immediately slow local rate limiter; poll /info userRateLimit; retry with capped exponential backoff + full jitter; consider opening circuit breaker if persistent.
HTTP 5xx / network error	Retry with capped exponential backoff + full jitter; count toward circuit breaker failure rate; do not let retries exceed attempt/time budgets.
Timeout on order submission	Mark as ambiguous; reconcile with /info orderStatus (by cload when available); do <i>not</i> blindly resubmit.
Repeated stale expiresAfter issues	Treat as a bug/incident; centralize expiry computation; stop aggressive retries because certain cancels may cost 5x rate-limit consumption.
Sustained failure cluster	Open circuit breaker to fast-fail; stop new order churn; focus on state repair and safe mode.

7.2.15 What to log and measure (so resilience is testable)

Resilience is only real if you can verify it under load and during incidents. At minimum, emit structured logs/metrics for:

- request counts by route (/exchange vs /info),
- HTTP status distributions (especially 429 and 5xx),
- retry counts and total retry time per command,
- number of ambiguous outcomes and time-to-reconciliation,
- circuit breaker state transitions and time spent open,
- local rate limiter utilization and userRateLimit utilization from /info.

Notice what is missing: none of this requires knowing *why* the exchange was slow. You only need symptom-level signals that your sys-

tem can observe and act on. That keeps the design robust across changing network conditions and platform behavior.

7.2.16 Putting it together: safe behavior under stress

A well-designed execution layer behaves conservatively when external reliability degrades:

- it *reduces* order churn as rate-limit pressure rises,
- it *backs off* with jitter instead of piling on,
- it *stops* and fast-fails when persistent failure suggests that retries will do more harm than good,
- it *reconciles* ambiguous outcomes using `/info orderStatus` rather than guessing.

This is not just engineering hygiene. In perps, the highest-cost incidents are rarely “the strategy was wrong.” They are “the system behaved unpredictably when the market was chaotic.” Deterministic timeouts, bounded retries, adaptive throttling via `userRateLimit`, and operational circuit breakers are the toolkit that keeps the bot’s behavior understandable when you need it most.

7.3 System Latency Optimization

After you decouple strategy, risk, and execution and harden the system against rate limits and partial failures, the next constraint is speed. On Hyperliquid perps, speed is not a vanity metric: the value of a quote or taker decision can decay within milliseconds, and the cost of being late often appears as adverse selection (you trade *after* the information moved the price).

This section focuses on *tick-to-trade* latency: the elapsed time from a market-data update arriving at your bot to the moment the corresponding order action is submitted and acknowledged. The objective is not

“minimize average latency”; it is to control **tail latency** and jitter while preserving correctness and safety.

7.3.1 What to optimize: end-to-end latency, not a single stopwatch

Latency in production is a pipeline. If you only time one segment (e.g., HTTP round-trip), you can improve that number while making the overall system worse (e.g., by introducing queueing delays elsewhere).

A practical, measurable decomposition (using timestamps captured in code) is:

Segment	Meaning
$t_{ws\_recv} \rightarrow t_{decoded}$	WebSocket receive to parsed/validated event in internal format.
$t_{decoded} \rightarrow t_{signal}$	Event available to alpha, features computed, decision made.
$t_{signal} \rightarrow t_{order\_built}$	Intent converted to order command (including risk checks).
$t_{order\_built} \rightarrow t_{signed}$	Signing and final payload construction.
$t_{signed} \rightarrow t_{submitted}$	Queued to execution and written to the network stack.
$t_{submitted} \rightarrow t_{ack}$	Exchange response received (or local timeout).

Keep the same decomposition in backtests/simulations when possible. Even if the absolute numbers differ, the breakdown tells you where time is being spent and which changes actually reduce tick-to-trade.

7.3.2 Use distributions, not averages

Monitoring must consider distributions/tails (not only averages) to avoid missing high-percentile latency regressions. For trading, p50 is “how fast you are when things are easy.” p95/p99 is “how fast you are when the market is hard,” which is exactly when execution quality matters most.

At minimum, record and trend:

$$\{p50, p90, p95, p99\}$$

for each segment and for end-to-end tick-to-trade.

7.3.3 Tail latency: why it dominates and where it comes from

Tail latency is driven by hardware/OS/application factors beyond pure queueing models; contributors include background interference, scheduling/concurrency models, interrupt routing, CPU power-saving, and NUMA effects. This is the main reason latency feels “random” in production even when your code path is deterministic.

For trading bots, the implications are:

- **A small amount of jitter can erase edge.** A strategy designed around a 5 ms reaction time can become uncompetitive if it occasionally reacts in 50 ms.
- **Adding threads may increase tails.** More concurrency can mean more context switches, lock contention, cache misses, and GC pressure.
- **Host configuration matters.** The same binary can behave very differently on a noisy shared host versus a dedicated tuned machine.

This does not mean you need “HFT infrastructure” to trade successfully; it means you must avoid architectural choices that *create* tails unnecessarily.

7.3.4 Concurrency model choices: keep fast paths uncontended

In the previous sections, decoupling gave you asynchronous components. Latency optimization asks a further question: *how many things should run concurrently, and how do they share state?*

7.3.5 Single-writer state: a simple rule that prevents locks

One component is the single writer of authoritative state; everyone else is a reader or receives snapshots.

This reduces or eliminates locks. Locks are not always slow on average, but they are notorious tail-latency generators under contention.

A common implementation is:

- one state thread/task applies events sequentially,
- strategy and risk read an immutable view (copy-on-write snapshot, versioned pointer, or message-passed state),
- execution does not mutate strategy state; it emits results back as events.

The trade-off is that you must think carefully about when a view is “fresh enough.” That is a strategy design choice, not a concurrency accident.

7.3.6 Event loop vs thread pool: choose based on blocking behavior

For most Hyperliquid bots, the dominant latencies are network and queueing, not raw CPU. That pushes you toward an **event loop** model:

- WebSocket ingestion and HTTP submission are asynchronous,
- CPU work is kept short and predictable,
- long work is offloaded explicitly (so it cannot block the loop).

Use a thread pool only for workloads that are truly blocking or heavy (e.g., large data transforms, compression, heavy cryptography), and even then, cap concurrency to avoid CPU saturation and scheduler jitter. In trading, “more parallel” often becomes “more variance.”

7.3.7 Backpressure is also a latency tool

Queue growth is latency growth. If your ingestion queue lengthens, the time between *event arrival* and *event processing* increases even if your compute is fast. That delay is part of tick-to-trade.

Treat queue depth and event lag as first-class latency metrics:

$$event_{lag} = t_{now} - t_{ws,ecv}$$

If event lag exceeds a threshold, it is often better to:

- reduce order churn,
- widen quoting parameters,
- or temporarily pause trading decisions,

rather than to keep sending orders based on stale information.

7.3.8 Measurement infrastructure: make latency attributable

If you cannot localize latency, you cannot optimize it. This is where tracing becomes practical, not decorative.

OpenTelemetry supports tracing across process/network boundaries; correlated traces can localize latency to components (WS decode, strategy compute, signing, HTTP submit) and separate compute vs network delays.

A disciplined tracing setup for a bot uses:

- one trace/span per “decision cycle” (e.g., per market event that triggers evaluation),
- nested spans for decode, feature compute, risk check, signing, and submit,
- consistent correlation IDs (e.g., CLOID) attached to spans and logs.

Two design cautions:

- **Sample carefully.** Full tracing on every tick can be expensive; use sampling or trigger detailed traces during incidents.
- **Measure overhead.** Observability can increase latency if implemented naively (synchronous logging, heavy serialization).

7.3.9 Network-path latency: control what you can, instrument the rest

Even perfect code cannot beat physics and routing, but there are still meaningful improvements available.

7.3.10 TCP behavior: small messages and Nagle’s algorithm

TCP-level optimizations matter for latency-sensitive applications: Nagle’s algorithm trades latency for bandwidth efficiency; `TCP_NODELAY` is commonly used to avoid delay for small messages.

For trading bots, many messages are small (order submissions, cancels, heartbeats). If Nagle’s algorithm coalesces small writes, you can incur extra delay. Setting `TCP_NODELAY` for critical low-message-size control paths is a common approach, but the key is to **verify with instrumentation rather than assuming**. Depending on your HTTP client and runtime, the setting may or may not affect the relevant sockets.

7.3.11 Connection reuse and warm paths

Although the exact client implementation varies, the principle is stable:

- avoid repeated connection setup,
- keep critical routes “warm” (ready to send),
- separate market-data sockets from trading-action sockets so a burst in one channel does not stall the other.

This is consistent with the decoupled architecture: multiple I/O loops can exist, but they should not compete for the same locks or queues.

7.3.12 Host and runtime tuning: reduce jitter before chasing microseconds

Because tails are often host-driven, a short checklist can yield larger improvements than low-level code tweaks.

Host tuning ideas consistent with tail-latency research include:

- **Pin processes/threads** to reduce scheduler migrations.
- **Minimize noisy neighbors** (prefer dedicated instances/cores when possible).
- **Disable CPU power saving** for dedicated trading hosts, since frequency scaling and deep sleep states can add wake-up latency.
- **Avoid cross-NUMA memory access when possible**, because remote memory access adds variable delay.

You do not need all of these to trade, but you should know which ones you have *not* controlled so you can interpret your p99 latency correctly.

7.3.13 Batching and throttling: latency versus control

Batching can improve throughput but can increase latency. The right choice depends on your trading style.

When batching helps.

If your strategy generates many small, related actions (e.g., updating a ladder of quotes), batching can reduce per-request overhead and rate-limit consumption. It can also reduce lock contention inside your own bot by grouping work.

When batching hurts.

If you batch by waiting for “more items,” you introduce intentional delay. For latency-sensitive actions (risk-off cancels, fast taker hedges), do not wait.

A common compromise is **micro-batching**:

- batch within a very small time window (e.g., a few milliseconds),
- flush immediately on critical signals (inventory breach, kill-switch condition),
- measure whether the reduction in overhead offsets the added waiting time.

7.3.14 Latency and execution quality: optimize the right objective

Lower tick-to-trade latency is not automatically better. Faster quoting can increase adverse selection if it causes you to update quotes aggressively into toxic flow, or if it encourages excessive churn that triggers rate-limit pressure. The correct objective is **net execution quality after costs**:

- fills at acceptable prices,
- controlled inventory,
- stable behavior under volatility,
- minimal wasted order traffic.

This is why measurement must include both:

- latency distributions (p95/p99 tick-to-trade),
- and outcome metrics (fill rates, slippage/adverse selection proxies, order-to-fill ratios).

7.3.15 A step-by-step optimization workflow

A rigorous latency program is iterative and evidence-driven:

- **1. Instrument** the full latency breakdown and queueing delays.
- **2. Baseline** p50/p95/p99 for each segment during representative market periods.

- **3. Localize** the dominant contributor using traces (OpenTelemetry) and segment metrics.
- **4. Change one thing** (concurrency cap, queue design, host setting, socket option).
- **5. Re-measure** in the same conditions; verify p95/p99 improved, not just p50.
- **6. Check trading outcomes** to ensure you did not “win the stopwatch” while losing execution quality.

This workflow keeps latency engineering grounded. In production trading, the easiest mistake is to chase a micro-optimization that looks good in isolation but increases jitter, reduces safety, or worsens fills. By treating tick-to-trade as a decomposed, observable pipeline and focusing on tails, you can improve speed without making the system brittle.

7.4 Testnet Validation and Simulation Workflows

7.4.1 The layered pipeline: unit tests, replay simulation, testnet acceptance, then mainnet

You want multiple gates, each catching a different class of bug:

- **Deterministic unit tests:** prove that your pure logic (state transitions, risk clipping, intent → command conversion) is correct.
- **Replay-based simulations:** feed recorded event sequences into the bot to validate that it behaves correctly under realistic message orderings and timing.
- **Testnet acceptance tests:** confirm that real WebSocket/HTTP interactions and exchange semantics match your expectations.

- **Mainnet rollout with minimal risk:** begin with tiny size and constrained permissions, and only then scale.

The reason to keep these layers distinct is practical: unit tests should run in seconds; replays may run in minutes; testnet can be slower and flaky; mainnet has real consequences. Each layer should be as fast and deterministic as possible.

7.4.2 Deterministic unit tests: making the bot testable by design

The event-driven architecture from earlier sections is what enables determinism. If alpha and risk are pure functions over a state snapshot, and if execution is abstracted behind an interface, you can test most trading logic without any network calls.

7.4.3 Test the state machine, not the network

Your state store should have a method that consumes an internal event and produces a new state (or updates in place). Unit tests should cover:

- snapshot hydration rules for user streams: ignore incrementals until a snapshot arrives; on `isSnapshot:true`, reset the relevant substate atomically,
- order lifecycle transitions as represented internally (open → partially filled → filled/canceled),
- idempotency: applying the same event twice should not double-count fills,
- handling of out-of-order messages: ensure the state machine either repairs safely or rejects inconsistent updates.

Even though Hyperliquid's user streams provide `isSnapshot:true` followed by incrementals, your code should be robust to reconnect-induced oddities (e.g., a late incremental arriving right after reconnect but before the new snapshot is processed).

7.4.4 Control time explicitly in tests

Time is the enemy of determinism. The solution is to inject time:

- replace calls to “now” with a Clock interface in modules that compute deadlines or `expiresAfter`,
- in unit tests, use a fake clock that you can advance deterministically.

This lets you test expiry handling and retry budgets without sleeping.

7.4.5 Floating-point hygiene: test rounding at boundaries

Most production trading bugs involving numbers occur at boundaries:

- size rounding that unexpectedly becomes zero,
- price rounding that crosses the spread,
- exposure checks that pass by 10^{-12} due to binary floating point.

Mitigation is not “never use floats”; it is to write tests that pin expected behavior at the edges. If your system uses decimal or integer tick units internally, test the conversions both ways (human input → internal units → order payload).

7.4.6 Replay-based simulation: validating behavior under realistic sequences

Unit tests answer: “Is my logic correct for hand-crafted cases?” Replays answer: “Does my logic remain correct under the messy event sequences the exchange produces?”

7.4.7 Build an append-only event log

A replay harness starts with recording. In production (and optionally on testnet), record an append-only log containing:

- raw WebSocket messages for market and user streams (including snapshot flags),
- timestamps at the moment of receipt (t_{ws_recv}),
- outgoing order commands and the time they were submitted,
- incoming HTTP responses and/or user-stream confirmations.

Do not record only “processed” state; record the inputs. The point is to reproduce the same sequence later.

A minimal log format is a line-delimited JSON where each line is one event with:

- type (market_ws, user_ws, http_request, http_response),
- ts_ns,
- payload.

7.4.8 Replays should be deterministic even when the original run was not

A good replay runner:

- re-emits events in recorded order,
- uses the recorded timestamps (or a scaled version) to reproduce inter-arrival timing,
- replaces real execution with a stub that reads recorded HTTP responses (or simulates them).

This is how you test the logic for ambiguous outcomes safely: you can feed the bot a timeout at /exchange, then later deliver a user-stream fill, and verify that reconciliation logic converges to the same final state.

7.4.9 Use replays to test stress behaviors

Replays are ideal for verifying safety behaviors that are hard to force reliably on a live exchange:

- WebSocket disconnect mid-fill, then reconnect with snapshot rebuild,
- bursts of 429 responses followed by recovery,
- delayed user events that arrive after a local timeout.

These are precisely the scenarios where a bot tends to overtrade, double-submit, or corrupt its order ledger if the state machine is slightly wrong.

7.4.10 Testnet validation: confirm real exchange semantics without risking capital

After your logic is stable in unit tests and replays, you still must verify that the integration points behave correctly. Hyperliquid provides a specific testnet WebSocket endpoint `wss://api.hyperliquid-testnet.xyz/ws`, enabling production-like WS ingestion testing without risking capital.

7.4.11 Environment separation: treat testnet as a first-class target

A production bot should run against:

- testnet URLs/endpoints in CI or staging,
- mainnet URLs/endpoints only in controlled releases.

The official Python SDK includes explicit testnet URLs/constants and example code, supporting CI-style integration tests that can run against testnet endpoints. Even if you do not use the SDK in production, this is a useful reference point: your own configuration

should make environment selection a single, obvious switch (and it should be hard to accidentally run `mainnet` from a developer laptop).

7.4.12 Test funds: operationalize funding like any other dependency

Operational testnet usage requires test funds (mock tokens); practical faucet guidance and cadence can be incorporated into onboarding/run-books.

Treat testnet funding as part of your operational readiness:

- document how to obtain and refresh test funds,
- include expected delays and rate limits,
- ensure the bot surfaces a clear error if the test account is unfunded (instead of failing deep in execution code).

7.4.13 Acceptance tests: the minimal end-to-end smoke suite

A useful testnet acceptance suite is intentionally small and repeatable. It should:

- Connect to `wss://api.hyperliquid-testnet.xyz/ws` and subscribe to the required market feeds.
- Subscribe to the user stream and verify snapshot-first behavior (`isSnapshot:true` then `isSnapshot:false`).
- Place a minimal-size order and verify it appears in user updates and is queryable through `/info orderStatus`.
- Cancel that order and verify the cancel is reflected in the same two channels (stream + `orderStatus`).
- Force a reconnect (intentionally drop the WebSocket) and verify that the bot rebuilds state from the next snapshot before trading resumes.

This suite is not about profitability. It is an integration contract test: the bot can observe, act, and reconcile in the same way it will on mainnet.

7.4.14 Chaos testing before mainnet: break your bot on purpose

If you only test the happy path, production will teach you the unhappy one. Chaos tests do not need to be complex; they need to be targeted:

- **Drop WebSocket** connections at random times (especially during open orders).
- **Delay HTTP** responses artificially (to trigger local timeouts and ambiguous outcomes).
- **Force 429 behavior** by lowering your local rate limits in test runs, ensuring that throttling and backoff logic triggers predictably.

The pass/fail criterion is not “no errors.” The criterion is: the bot converges back to a correct state and does not do unsafe things (double-submit, exceed limits, trade with stale state).

7.4.15 CI gates: making correctness the default

A hardened pipeline makes tests automatic. A pragmatic CI setup includes:

- **On every commit:** run unit tests and deterministic replay tests (fast).
- **Nightly or scheduled:** run testnet smoke tests (slower, external dependency).

Nightly testnet runs are especially valuable because they detect integration drift: if an endpoint behavior changes or your environment configuration breaks, you learn it before a mainnet incident.

To keep CI high-signal:

- keep testnet order sizes minimal,
- limit the number of symbols tested,
- enforce timeouts so CI does not hang on external flakiness.

7.4.16 Pre-deploy checklist: a short list that prevents most disasters

Before allocating meaningful capital, require an explicit checklist completion. A minimal version is:

- Testnet account funded and verified (runbook followed).
- WebSocket reconnect + snapshot rebuild validated on testnet.
- Order placement, cancellation, and `/info orderStatus` reconciliation validated on testnet.
- Chaos tests executed: WS drops, HTTP delays/timeouts, and forced throttling.
- Replay suite run on recent recorded data and passes deterministically.
- Metrics/tracing enabled (at least enough to debug latency segments and ambiguous outcomes).

7.5 Health Monitoring and Alerts

A production bot does not just *trade*; it operates continuously in an environment where failures are normal: WebSockets disconnect, HTTP requests time out, the host experiences latency spikes, and internal queues can silently accumulate. Monitoring is the system that tells you what is happening *now*, and alerting is the mechanism that forces a response when continuing to trade would be irresponsible.

This section focuses on building a real-time telemetry pipeline and an automated “kill switch” that can halt trading and cancel open orders

when the bot enters an unsafe state. The design philosophy is operational: detect symptoms quickly, avoid noise, and ensure every high-severity alert corresponds to an actionable runbook step.

7.5.1 Observability goals: detect symptoms early and explain them later

Alerting should minimize noise: pages are expensive; effective alerting is high-signal and action-oriented, and should generally focus on symptoms rather than causes. A good alert answers:

- **What is broken?** (symptom)
- **What is the impact on trading safety?**
- **What should the bot do automatically right now?**
- **What should a human do next (if paged)?**

Causes are still valuable, but they belong in dashboards and traces used during debugging. The practical ordering is: *stop the bleeding first, then investigate*.

A second design requirement is proactive instrumentation. The observability goal is to troubleshoot *unknown unknowns* without adding new instrumentation mid-incident; this implies careful upfront instrumentation design. In other words, if a new failure mode appears at 3am, your existing metrics/logs/traces must be enough to localize it.

7.5.2 Telemetry architecture: metrics, logs, and traces with correlation

OpenTelemetry provides a standard framework to export traces/metrics/logs and correlate them using context propagation, supporting incident debugging across services/components. Even for a single-process bot, the value of correlation is large because trading decisions span multiple asynchronous steps: WebSocket receive, decode, alpha compute, risk checks, signing, HTTP submit, and eventual order state reconciliation.

A robust telemetry stack uses all three signal types:

Metrics (time series). Low-cardinality, always-on measurements for alerting and trending (queue depth, error rates, p95/p99 latencies).

Logs (structured events). High-fidelity records of notable events (order rejected, reconnect, reconciliation mismatch). Keep them structured (JSON) so you can filter by symbol, CLOID/OID, and failure class.

Traces (end-to-end causal chains). Request-level spans that attribute latency and failures across components. Tracing is the fastest route to answering: “Is the slowness in WS decode, strategy compute, signing, or network submit?”

7.5.3 Standardize identifiers for correlation

Correlation only works if every layer uses the same identifiers:

- attach order identifiers (CLOID/OID) as attributes to logs and spans,
- propagate a “decision cycle” ID from market event to the resulting intents/commands,
- include a monotonic timestamp and a wall-clock timestamp in emitted telemetry.

This turns debugging from “grep and guess” into “filter and follow.”

7.5.4 What to monitor: a symptom-first checklist

The goal is not to measure everything. The goal is to measure the small set of quantities that, when unhealthy, imply unsafe trading.

Below is a practical symptom-oriented set of monitors that maps directly to earlier sections (architecture, resilience, and latency).

Symptom	Why it matters	Typical response
Order submit error rate	You cannot reliably change exposure or manage open orders.	Reduce activity; if sustained, trigger kill switch.
WS disconnected duration	State becomes stale; fills may be missed until snapshot-s/reconciliation.	Block new trading; reconnect; rehydrate state.
Stale market data age	Decisions are based on old information; fills become toxic.	Pause quoting/taking until data is fresh.
Position risk limit exceeded	Strategy or state is wrong; liquidation risk rises quickly in perps.	Immediate risk-off actions; possibly kill switch.
Order reconciliation backlog growing	Ambiguous outcomes accumulating; local state diverges from exchange truth.	Pause new orders; prioritize reconciliation/cancels.
Latency p99 regression	Tail latency destroys execution quality and may indicate host trouble.	Degrade modes; reduce churn; investigate host.

Notice that these are largely *symptoms*. You do not need to know whether the root cause is a bad route, a host CPU spike, or an upstream incident to decide that continuing to quote is unsafe.

7.5.5 Alert design: high signal, anti-flap, and runbooks

Two practical rules keep alerting sane:

High-signal only. Reserve paging for conditions that require immediate action. Everything else becomes a ticket or a dashboard indicator.

Prevent flapping. Flapping control is important: require alert conditions to hold for a minimum duration before firing to avoid transient noise. Concretely:

- require the threshold to be breached for multiple evaluation intervals,
- or apply a “for” duration (e.g., 30–120 seconds depending on metric).

7.5.6 Tie every page to a runbook

A page that does not have a defined next action is operational debt. Each critical alert should include:

- a one-line description of impact,
- a link to a dashboard filtered to the relevant symbol/account,
- a small runbook: “what to check first” and “how to put the bot in safe mode.”

7.5.7 Metamonitoring: monitor the monitoring

Prometheus alerting practices recommend symptom-based alerts and metamonitoring (ensuring the monitoring/alerting pipeline itself works, ideally via end-to-end tests). A simple approach is a periodic “synthetic alert” that should always traverse the chain: Prometheus → Alertmanager → notification channel. If the synthetic check fails, you alert on the alerting system itself. This is unpleasantly recursive, but necessary: a silent monitoring failure during a market event is one of the worst operational states.

7.5.8 Automated kill switch: design goals and safety properties

The kill switch is a control plane that can take over from strategy when safety is compromised. Its job is not to be clever; its job is to be decisive and predictable.

A good kill switch has the following properties:

- **Fast:** it must activate quickly when a dangerous symptom triggers.
- **Conservative:** in doubt, reduce exposure and stop adding risk.
- **Idempotent:** repeated activation should not create new risk or new orders.
- **Independent:** it should not rely on the same fragile pathways that might be failing (e.g., do not require the alpha engine to be healthy).
- **Observable:** every activation is logged, metered, and traceable.

The kill switch typically implements one or more actions:

- disable new order creation (block intents at the risk gate),
- cancel open orders,
- optionally enforce reduce-only behavior at the strategy/risk layer,
- escalate to human paging if unresolved.

7.5.9 Telemetry pipeline feeding the kill switch

The kill switch should not subscribe to every raw event. Instead, it should consume a small set of *health signals* computed from raw telemetry: disconnect duration, stale data age, error rates, queue depth, and position/margin breaches. This reduces complexity and keeps the decision logic auditable.

7.5.10 Decision engine: from signals to actions

Implement the kill decision as a clear policy function:

- inputs: current health signals + recent history window,
- outputs: trading mode (normal / degraded / halt) and required actions (cancel orders, stop new commands).

A useful pattern is to separate:

- **detection**: compute signals (metrics),
- **decision**: apply thresholds and hysteresis,
- **actuation**: send cancels and disable trading.

This separation makes it easier to test and avoids accidental coupling with strategy logic.

7.5.11 Actuation safety: do not create more failure load

Kill-switch actuation often happens during stress (rate pressure, slow network). To keep it safe:

- prioritize a minimal set of critical cancel actions,
- rate-limit cancels and apply bounded retries (as discussed earlier),
- if cancellations cannot be confirmed quickly, shift to reconciliation mode and stop trading until state is coherent.

7.5.12 Kill-switch schematic

7.5.13 Concrete kill-switch triggers (examples)

Below are examples of triggers that are easy to implement and test. The specific thresholds depend on your strategy and risk appetite, but the structure should remain consistent.

Connectivity trigger. If WebSocket is disconnected for longer than a threshold, halt new trading. Rationale: state is stale, and fills/cancels may be missed.

Stale market data trigger. If the age of the latest market event exceeds a threshold (e.g., event lag grows beyond acceptable bounds), stop quoting/taking. Rationale: late decisions are often worse than no decisions.

Execution failure trigger. If order submit error rate rises above a threshold for a sustained window, open the kill switch. Rationale: inability to place/cancel implies inability to control exposure.

Risk breach trigger. If position/inventory exceeds a hard limit or if a safety invariant is violated (e.g., exposure grows while the bot believes it is flat), immediately cancel orders and halt. Rationale: this is a direct liquidation-risk pathway.

Reconciliation backlog trigger. If the number of “uncertain” orders (ambiguous outcomes awaiting /info reconciliation) grows and

does not clear, stop new orders. Rationale: when state is uncertain, additional trading compounds uncertainty.

7.5.14 Testing the monitoring and kill switch

Because the kill switch is a safety mechanism, you should test it as if it were trading logic:

- **Unit tests:** feed synthetic health signals into the decision engine and verify state transitions (normal → degrade → halt, with hysteresis).
- **Replay tests:** inject sequences like disconnect → reconnect → snapshot, verifying the bot halts and then resumes only when safe.
- **End-to-end tests:** verify that kill-switch actuation actually blocks new order commands and issues cancels through the execution engine.
- **Metamonitoring tests:** validate that an alert can traverse Prometheus→Alertmanager→notification channel; alert if the pipeline breaks.

The key operational stance is simple: if you cannot prove your monitoring works during calm periods, it is unlikely to work during volatility.

7.5.15 Operational discipline: keep the system understandable

A final warning: it is easy to accumulate dozens of low-quality alerts and hundreds of metrics. This creates alert fatigue and makes incidents harder to debug.

Keep a short, curated set of pages:

- pages for “unsafe to trade” symptoms,
- tickets for degradation that can wait,

- dashboards and traces for root cause analysis.

With that discipline, monitoring becomes part of the bot's control system rather than an afterthought: it detects unsafe states quickly, triggers automated containment via the kill switch, and preserves enough telemetry to explain what happened after the fact.

7.6 Key Management and Production Security

The previous sections established how to make a bot correct, resilient, and observable. None of that matters if an attacker can sign transactions or trading actions as you. In on-chain perps, the private key is not merely a login credential; it is a *signing authority* that can move capital and place trades. Hyperliquid's SDK usage patterns require handling private keys for authentication/trading; production systems must therefore design key isolation, rotation, and incident response around those keys.

This section focuses on institutional-grade practices: least privilege, secure secret storage and injection, runtime protections, and a realistic incident response posture.

7.6.1 Threat model first: defend against how keys are actually stolen

Security design improves dramatically when it is grounded in attacker behavior rather than abstract "best practices." A vulnerability/incident-driven view of wallets helps prioritize controls for real attacker behavior (phishing, key exfiltration, compromised hosts, supply-chain risks). For bot operators, the dominant threats are typically:

- **Key exfiltration from disk:** plaintext keys in config files, home directories, CI artifacts, or container layers.
- **Key leakage through telemetry:** keys accidentally printed to logs, crash dumps, metrics labels, or tracing attributes.

- **Compromised runtime:** malware, compromised dependencies, or an attacker with shell access reading process memory.
- **Operator phishing:** tricking a human into authorizing a malicious key export or signing action.
- **Supply-chain risks:** malicious packages, compromised build pipelines, or poisoned Docker images.

You cannot eliminate all risk, but you can choose where the key lives, who can request signatures, and how quickly you can revoke that authority.

7.6.2 Principle of least privilege: reduce what any single key can do

As we saw earlier, separating duties is a foundational control. Here we extend that idea into concrete operational boundaries.

7.6.3 Two-tier key model: trading hot key vs treasury cold funds

A simple, high-leverage control is to separate:

- a **hot trading key** used by the automated bot, funded only with the capital required for the strategy's risk limits, and
- a **treasury/cold wallet** holding the majority of funds, not accessible to the bot runtime.

This reduces the maximum loss from a full bot compromise. It also changes incident response: you can rotate or revoke the hot key without disrupting long-term custody.

Operationally, enforce that any transfer from treasury to hot key is:

- explicit (manual approval or multi-party approval),
- monitored (alerts on transfers),
- rate-limited (no automated "top up" loops).

7.6.4 Privilege by environment: testnet keys must never touch mainnet

Treat environment separation as a security boundary:

- separate testnet and mainnet keys,
- separate vault paths and access policies,
- separate CI secrets and runtime secrets.

This prevents a class of operational accidents where a developer or CI job accidentally deploys a mainnet-capable key into a non-production environment.

7.6.5 Secure secret storage and injection: do not ship keys with code

The primary rule is simple:

A production key should not be present in source control, container images, build artifacts, or static configuration files.

Instead, keys are either (i) stored encrypted at rest and injected at runtime, or (ii) never exposed to the runtime at all, with signing delegated to a service.

Two facts are particularly useful for institutional designs:

- Vault Transit enables cryptographic operations (including signing/verification) without exposing raw key material to application runtimes, supporting least-privilege designs.
- OpenZeppelin Defender Relayers illustrate a KMS-backed pattern where private keys are generated/stored in KMS and signing happens inside KMS; API keys can be rotated separately—an operationally relevant separation of duties.

These are examples of the same structural idea: isolate signing authority behind an authenticated, audited interface.

7.6.6 Architecture: encrypted vault injection into a restricted runtime

The following architecture keeps secrets in an encrypted store and injects them only into a constrained environment, with explicit boundaries and audit points.

This pattern is intentionally explicit about *where* secrets can exist:

- encrypted at rest (vault),
- briefly in memory (restricted runtime),
- never in logs or on disk.

7.6.7 Signing isolation patterns: local hardened host vs delegated signing

There are two common deployment patterns, both viable depending on team maturity and latency requirements.

7.6.8 Pattern A: local signing on a hardened host

The bot process holds the key in memory and signs locally. This can be acceptable if you can enforce:

- dedicated host/VM (reduce attack surface and noisy neighbors),
- strict OS user isolation (bot runs as non-root),
- minimal filesystem permissions (no writable directories except for necessary state),
- no key persistence (key injected on startup, not written to disk),
- careful crash handling (disable core dumps; sanitize exception traces).

The primary downside is that a runtime compromise is a key compromise.

7.6.9 Pattern B: delegated signing via Vault Transit / KMS-backed service

In this model, the bot never sees raw key material. Instead it submits a signing request to a service that enforces policy and writes audit logs. Vault Transit enables cryptographic operations (including signing/verification) without exposing raw key material to application runtimes, supporting least-privilege designs. OpenZeppelin Defender Relayers illustrate a KMS-backed pattern where private keys are generated/stored in KMS and signing happens inside KMS; API keys can be rotated separately—an operationally relevant separation of duties.

The benefits:

- compromise of the bot process does not automatically yield the private key,
- signing can be gated by policy (symbol allowlists, size limits, time windows),
- every signature request is auditable.

The trade-off is latency and additional operational components. For many strategies, the added latency is acceptable relative to the security gain, but you must measure it and decide deliberately.

7.6.10 Secrets handling in practice: the “never leak” checklist

Most key compromises in bot environments are accidental. The following controls prevent common leakage paths:

No secrets in logs/metrics/traces.

- Do not log request payloads that might contain sensitive fields.
- Do not attach secrets as OpenTelemetry span attributes.
- Treat “debug logging” as production code with production consequences.

No secrets in crash artifacts.

- Disable core dumps in production where possible.
- Ensure exception handlers sanitize context.
- Keep stack traces but redact sensitive objects.

No secrets in build pipeline.

- CI should never have access to production signing keys.
- Container images should not contain secrets in layers or environment defaults.

Short-lived credentials for secret access. Even if the signing key itself is long-lived, access to retrieve it (or access to request signatures) should be short-lived and revocable. This reduces the damage from stolen tokens.

7.6.11 Runtime protections: reduce the blast radius of compromise

Monitoring and kill switches reduce loss during system faults; security controls reduce loss during hostile faults.

Practical runtime hardening includes:

- **Network egress control:** allow outbound traffic only to required endpoints; block arbitrary egress that could exfiltrate secrets.
- **Dependency pinning and provenance:** lock versions, validate hashes where possible, and review dependency changes.
- **Principle of minimal permissions:** the bot should not have access to other services, files, or credentials it does not need.
- **Memory discipline:** keep key material in as few objects as possible; overwrite/zeroize buffers when done.

These are not exotic techniques; they are about removing easy wins for attackers.

7.6.12 Incident response playbook: assume compromise is possible

Institutional-grade security is not "perfect prevention." It is the ability to respond quickly and predictably.

A minimal playbook includes:

7.6.13 1) Containment: stop signing and stop trading

- Trigger the operational kill switch to halt new orders and cancel open orders.
- Revoke the bot's ability to access signing (disable Vault policy/token, revoke API key to signing service, or shut down the relayer).

7.6.14 2) Key rotation: replace compromised authority

- Generate a new hot trading key.
- Update vault/KMS policies and bot configuration to use the new key.
- Keep the old key monitored for unexpected activity until fully retired.

7.6.15 3) Forensics: preserve evidence without leaking secrets

- Snapshot relevant logs, traces, and audit records.
- Do not copy memory dumps unless you have a secure forensic process; they may contain sensitive material.

7.6.16 4) Restore: resume in constrained mode

Restart trading with:

- minimal balances,
- conservative limits,
- enhanced alerting on signing requests and transfers.

This mirrors how you would roll out a new strategy: small, observable, reversible.

7.6.17 Reasoning about wallet security systematically

Wallet security can be reasoned about systematically via authentication factors and wallet categories, including modern approaches such as threshold signatures and centralized co-signing. In practice, this means you should be explicit about:

- **who** can request a signature (bot only, bot + human approval, multi-party),
- **where** signing occurs (local process memory, Vault Transit, KMS-backed relay),
- **what** is allowed to be signed (policy limits),
- **how** actions are audited (immutable logs, alerting on anomalies).

These are design choices that can be evaluated like any other part of a trading system: by analyzing failure modes, measuring operational overhead, and selecting the least risky approach that still meets execution requirements.

7.6.18 Summary: the security posture that survives real operations

A production-grade Hyperliquid bot should assume that:

- keys are the highest-value target,
- attackers exploit operational mistakes more often than cryptographic weaknesses,
- strong separation (hot vs cold, testnet vs mainnet, bot vs signer) is more effective than ad-hoc hardening.

By combining least privilege, secure vault-based secret handling (or delegated signing), careful runtime protections, and a practiced incident response, you turn key management from a hidden single point of failure into a controlled subsystem with clear boundaries and predictable behavior.

