ChatGPT

Feature Mapping and Implementation Plan for dmorazzini23/ai-trading-bot

Executive Summary

Steps taken (access + analysis workflow): - Confirmed enabled connector availability via `api_tool` and used the **GitHub** connector to inspect repository artifacts (docs, configuration, and core modules). - Opened and analyzed the attached report as a **local PDF** at `/mnt/data/Institutional-Grade Trading Bots_ Decision Behavior, Profit Drivers, and a Gap Analysis Against Your.pdf` (8 pages) using local parsing (page-referenced excerpts included where relevant). - Mapped report features (decision pipeline, data requirements, risk controls, execution, TCA, learning loop, observability, and durability) to repository modules and docs. - Identified gaps (missing/partial features) and translated them into concrete, testable implementation tasks. - Consulted primary/official sources for key referenced concepts (market access risk controls, IEX vs SIP data, live vs backtest order-event behavior, optimal execution framing, and rate limits). [1]

Enabled connectors (as requested): - github

Bottom line: - The attached report's core framing—**institutional-grade bots as a decision pipeline** (alpha → risk → execution → post-trade learning) and "net-of-cost" governance— **maps strongly** onto this repository's documented architecture and many implemented modules. - The most material remaining deltas are **durability and "true parity" realism**: a **durable intent/order store with exactly-once semantics**, a **replay/backtest harness that mimics asynchronous live order events**, and deeper **execution-policy selection + calibration** to fully realize the TCA-first philosophy. The report explicitly calls these out as the "biggest remaining gap." (Report pages 4–7, by section headings.)

Research Approach

Report access notes - File format: **PDF**. - Accessibility: readable locally from the mounted path above. - Tooling limitation: the `file_search` tool did not expose the uploaded PDF for line-based `filecite` citations in this session; therefore, I reference **page numbers and section headings** from the extracted PDF text.

Repository access notes - Repository analyzed: `dmorazzini23/ai-trading-bot` (and only this repo). - Repository inspection used the GitHub connector to pull docs and code modules directly.

External sources policy - For concepts the report explicitly references (market access risk management, SIP vs IEX data provenance, asynchronous order events, and optimal execution), I relied on: - U.S. Securities and Exchange Commission [2] Rule 15c3-5 materials [3] - Alpaca Markets [4] documentation on IEX vs SIP feeds and API limits [5] - QuantConnect [6] LEAN docs describing live vs backtest order-event timing differences [7] - Optimal execution framing from Robert Almgren [8] and Neil Chriss [9] (Journal of Risk

publication pages) [10] - Implementation shortfall benchmark framing (Perold/Treynor lineage) via a trade-cost text excerpt and an industry TCA discussion [11]

Findings From the Attached Report

What the report defines as "institutional-grade"

The report's central claim is that "institutional-grade" does **not** mean "one ML model predicting up/down," but rather a **decision pipeline** that (a) constrains the tradable universe, (b) generates probabilistic edge, (c) converts edge into risk-budgeted targets, (d) enforces pre-trade controls, (e) executes with cost-awareness, and (f) learns from realized fills/slippage/opportunity costs (a tight feedback loop). (Report page 1, section "A clean institutional decision loop looks like this.")

This decomposition matches common production-system separations of: - **Alpha vs risk vs execution**, and - **Research vs live governance**, with explicit parity constraints (Report pages 1 and 4).

Key features, algorithms, and implementation expectations

The report enumerates several "institutional" behaviors and concrete controls:

- **Universe + eligibility gates**: liquidity/volatility constraints; (for shorts) borrowability; and strict **data-quality gating** to avoid "false edge" and survivorship pitfalls. (Report page 1.)
- **Signal generation**: probabilistic forecast/edge with explicit horizon and decay; signals are uncertain and must clear cost thresholds. (Report page 1.)
- **Portfolio construction**: risk budgets, correlation constraints, and sizing rules converting signals to targets. (Report page 1.)
- **Pre-trade risk controls and circuit breakers**: max order dollars/shares, price collars, order/cancel rate limits, kill switches, "cancel all on kill." The report connects these to market-access risk-control principles. (Report page 3 discussion of "market access rule style guardrails.") The cited regulatory archetype is U.S. Securities and Exchange Commission [2] Rule 15c3-5, which requires risk controls reasonably designed to prevent erroneous orders and limit exposure, under broker-dealer control, with regular review. [3]
- **Execution quality with TCA-first evaluation**: emphasizes **implementation shortfall** and "arrival benchmark" framing, not just P&L; encourages execution-policy selection (passive vs aggressive vs slicing) tied to spread/volatility/liquidity and participation limits. (Report pages 3–5.) Implementation shortfall is widely used as a benchmark comparing "paper" decision price vs actual realized execution (and may include delay/impact/opportunity cost depending on convention). [11]
- **Optimal execution framing**: references the classic tradeoff between impact costs and risk/uncertainty and points toward the Robert Almgren [8] –Neil Chriss [9] optimal execution perspective (cost frontier; cost vs risk tradeoff). [10]
- **Data realism and microstructure**: the report warns that execution modeling and TCA credibility depend on quote quality and data provenance (notably IEX-only vs SIP/NBBO-grade coverage). It argues for explicit regimes when operating on "degraded" data. (Report page 4.) Alpaca Markets [4] 's own documentation distinguishes free IEX-based data from SIP feeds and documents SIP vs IEX stream endpoints. [12]
- **Live/backtest parity contract**: the report calls for an event-driven simulation/replay mode that mimics asynchronous order events (fills between bars), emphasizing that live trading order events

can arrive asynchronously even if the strategy runs on coarse bars. This matches QuantConnect [6]'s description that backtests trigger order events synchronously, while live order events occur asynchronously from brokerage threads. [7]

- **Durable state + auditability**: beyond JSONL logs, institutional OMS designs separate immutable audit logs from queryable state stores and run reconciliation after restarts. Key phrase: **"exactly-once intent" semantics** (persist intent ID → execute → reconcile on restart). (Report pages 5–6.)
- **Research governance and robustness**: parameter sensitivity sweeps; stability criteria; and a signed model/parameter manifest (data sources, embargo rules, cost model version, training window) for reproducibility. (Report page 5.)
- **Observability**: health endpoints + Prometheus are good, but institutional readiness adds alert routing, runbooks, quarantine/rollback triggers when slippage/reject rates spike. (Report pages 5–6.)

Roadmap proposed by the report

The report groups work into three phases (Report pages 6–7): - **Reliability and safety hardening first**: durable state store, crash-safe reconciliation, verified kill switch and caps, strict degraded-data regimes. - **Bring research harness closer to live reality**: streaming backtests, async fills, realistic impact/slippage and corporate actions/survivorship defenses, walk-forward gates, and calibrating cost models from your own fills. - **Execution quality and adaptive policy last**: execution strategy selection model, liquidity-regime classification feeding sizing/execution, continuous TCA rollups separating alpha from execution.

Repository Analysis

High-level structure and components observed

Based on repository docs and code modules pulled via the GitHub connector, the repo is organized as an engineering system with: - A core Python package `ai_trading/` (data pipeline, execution engine, risk tooling, monitoring/health). - Extensive documentation artifacts: `ARCHITECTURE.md`, `INSTITUTIONAL_PLATFORM.md`, `DEPLOYMENT.md`, `TESTING_FRAMEWORK.md`, and domain docs such as `docs/rate_limits_and_slos.md`. - Runtime/deploy tooling: `pyproject.toml`, `requirements.txt`, `DEPLOYMENT.md` deployment recipes (systemd/Docker/Kubernetes), and scripts (e.g., health checks). - A non-trivial test suite described in `TESTING_FRAMEWORK.md`, including deterministic test guidance and distinct unit/integration/slow categories.

Dependencies and integration points

The deployment guide and docs indicate a broker integration focus on Alpaca Markets [4] (alpaca-trade-api runtime pin and alpaca-py for tooling/tests in the deployment doc). [13]

The repo's own "rate limits and SLOs" documentation describes: - A **token bucket** style central rate limiter (burst capacity + refill rate) for different routes (orders/bars/quotes/etc.). - SLO thresholds for latency, turnover, data staleness, and drift, with circuit breaker hooks.
This aligns strongly with the report's governance/operations emphasis and also matches Alpaca Markets [4] API throttling realities (e.g., 429 on exceeded limits; the published baseline is 200 requests/minute per account for trading APIs). [14]

Execution, risk, and durability signals

From inspected modules: - The repo includes an execution engine (`ai_trading/execution/engine.py`), execution algorithms (`ai_trading/execution/algorithms.py`), idempotency support (`ai_trading/execution/idempotency.py`), reconciliation (`ai_trading/execution/reconcile.py`), and slippage logging (`ai_trading/execution/slippage_log.py`). - Risk tooling exists (e.g., `ai_trading/risk/manager.py`, `ai_trading/risk/kelly.py`) consistent with volatility/drawdown/position-sizing governance. - A database layer is present but appears **incomplete/placeholder** in places (e.g., `ai_trading/database/models.py`, `ai_trading/database/connection.py`), which directly corresponds to the report's callout that durable state is the key remaining gap (Report pages 5–6).

Testing and operational artifacts

The testing documentation describes deterministic runs (fixed seeds, env-cache refresh semantics) and references broad test coverage (unit/integration/slow), which aligns with the report's "parity contract + regression tests" recommendation. [7]

Feature-to-Code Mapping

Mapping table

Legend for Match Level - **Exact**: feature exists in repo with clear implementation and supporting docs/tests. - **Partial**: present but incomplete (stubs, in-memory only, insufficient parity realism, or missing calibration loop). - **Missing**: not found or only aspirational text without implementation hooks.

Report Feature	Repo File/Module (path)	Match Level	Notes
Decision pipeline separation (alpha → risk → execution → feedback)	`ARCHITECTURE.md`, `INSTITUTIONAL_PLATFORM.md`, `ai_trading/__init__.py` exports	Partial	Repo strongly documents modular separation; code appears to include the right modules, but end-to-end "contract tests" for parity are not clearly evidenced just from docs.
Universe/eligibility gating (liquidity/vol/data-quality)	`ai_trading/data/bars.py`, `ai_trading/data/fetch/…`	Exact	Data pipeline includes explicit health/gating semantics and multi-source normalization concepts; aligns with report's "eligibility gates."

Report Feature	Repo File/Module (path)	Match Level	Notes
Explicit degraded-data regime (IEX-only vs SIP/NBBO-grade)	`ai_trading/data/bars.py` + env-driven config; docs + monitoring	Partial	Repo supports degraded-mode behavior; report pushes for stronger "data provenance" invariants and stricter gating when feeds are degraded. Alpaca explicitly distinguishes IEX vs SIP feeds. [12]
Signal generation with horizon/decay and "edge clears costs" filters	`ai_trading/signals.py`, `ai_trading/alpha_decay.py`, `ai_trading/expectancy.py` (as suggested by exports/docs)	Partial	Report says your config already includes min expected edge, alpha decay, and anti-churn controls (Report page 3). The repo appears architected for this; verify end-to-end wiring.
Portfolio construction / risk budgeting	`ai_trading/risk/manager.py`, `ai_trading/risk/kelly.py`, (likely) `ai_trading/risk/engine.py`	Partial	Risk modules exist; "correlation constraints" and portfolio optimizer specifics are not clearly confirmed from inspected subset.
Pre-trade guardrails and circuit breakers (caps, collars, rate limits, kill switch)	`ai_trading/execution/engine.py`, monitoring + config, `docs/rate_limits_and_slos.md`	Exact	Strong alignment; this maps to the intent of SEC market access risk controls (limit exposure; prevent erroneous orders). [3]
Central rate limiting (avoid 429; per-route budgets)	`docs/rate_limits_and_slos.md`, `ai_trading/integrations/rate_limit.py` (per doc)	Exact	Matches Alpaca's published throttling behavior and the report's mention of tuned knobs. [13]
Health endpoints + Prometheus metrics	`ai_trading/health.py`, `ai_trading/health_monitor.py`, `ai_trading/monitoring/order_health_monitor.py`, `DEPLOYMENT.md`	Exact	Repo includes explicit health checks and monitoring guidance; report recommends adding alert routing/runbooks beyond metrics.

Report Feature	Repo File/Module (path)	Match Level	Notes
TCA and implementation shortfall framing (arrival benchmark)	`ai_trading/execution/slippage_log.py` + related TCA reports	Partial	Logging exists; report recommends consistent benchmark definition and systematic rollups including opportunity cost. Implementation shortfall framing is standard. [11]
Execution algorithms (TWAP/VWAP/POV + choices)	`ai_trading/execution/algorithms.py`	Partial	Algorithms exist, but report recommends liquidity/urgency-aware selection and systematic calibration (optimal execution framing). [10]
Execution policy selection model (passive vs aggressive vs slice based on spread/vol/liquidity)	(Not clearly isolated) likely inside `ai_trading/execution/engine.py`	Missing	Report explicitly asks to make "execution policy selection a first-class model" (Report page 5).
Live/backtest parity via async order events	`ai-offline-replay` entry point (per `pyproject.toml`), replay-related modules (implied), tests	Partial	Repo has replay tooling and strong test guidance; however the report asks to explicitly replay asynchronous live order events. QuantConnect documents this exact mismatch in live vs backtest order-event timing. [7]
Durable state store (OMS ledger + decision records in Postgres)	`ai_trading/database/models.py`, `ai_trading/database/connection.py`	Partial	Present but appears stub-like; report treats this as a major gap (Report pages 5–6).
Exactly-once intent semantics (persist intent → execute → reconcile after restart)	`ai_trading/execution/idempotency.py`, `ai_trading/execution/reconcile.py`	Partial	Idempotency exists but appears in-memory; durable intent store is needed for restart safety.

Report Feature	Repo File/Module (path)	Match Level	Notes
Parameter sensitivity sweeps + stability gates	(Not clearly present)	Missing	Report recommends small-grid perturbation sweeps and “stability criteria” (Report page 5).
Model / parameter registry with signed manifests (data window, embargo, cost model version)	(Not clearly present)	Missing	Report recommends a registry/manifest for reproducibility and governance (Report page 5).
Alert routing + automated quarantine/rollback	SLO doc describes circuit breakers; concrete notification integrations unclear	Partial	Core monitoring exists; report asks for alert routing and operational runbooks tied to deviations (Report page 6).

Repo components not emphasized in the report

The repo appears to contain broader experimentation infrastructure (for example, optional dependency groups involving meta-learning / reinforcement learning referenced in testing docs). The report focuses on **execution realism, durability, and governance** rather than expanding the strategy/model universe—so these advanced-model components are “extra” relative to the report’s scope.

Gaps and Concrete Implementation Tasks

Missing or partial features the report prioritizes

Durable intent and OMS state store

Why it matters: The report calls out durability as the biggest gap: JSONL logs are good but insufficient for restart-safe reconciliation and learning; institutional systems separate immutable logs, queryable state, and reconciliation (Report pages 5–6).

Implementation tasks - Implement a real SQLAlchemy-backed Postgres store (models + sessions) for: - `Intent` (idempotency key, decision timestamp, target qty, side, strategy, expected edge estimate, regime) - `Order` (broker order id, client order id, status transitions) - `Fill` (qty, price, timestamp, fees) - `DecisionRecord` (features snapshot hash, model version, cost model version) - Optional `PositionSnapshot` for reconciliation. - Modify execution engine to enforce: - **Persist intent first** (transaction commit), then submit order(s). - On startup: load “open intents” and reconcile against broker state. - Move idempotency storage from in-memory TTL to DB-backed keys.

Complexity: High
Suggested files - Modify: `ai_trading/database/models.py`, `ai_trading/database/`

`connection.py`, `ai_trading/execution/engine.py`, `ai_trading/execution/idempotency.py`, `ai_trading/execution/reconcile.py` - Add: `ai_trading/oms/intent_store.py`, `ai_trading/database/migrations/` (or a minimal migration script)

Enforced live/backtest parity with asynchronous order events

Why it matters: The report explicitly demands an event-driven replay that models fills between bars (Report page 4). This is a known source of backtest/live divergence; QuantConnect [6] documents that backtests fire order events synchronously while live fills arrive asynchronously from brokerage threads. [7]

Implementation tasks - Create a "simulation broker adapter" that: - Accepts orders from the same OMS interface used in live trading. - Emits order events asynchronously (e.g., via an async task queue) while strategy consumes market bars. - Update offline replay entry point to: - Replay market data + generate order-event timelines. - Produce decision records and enforce invariants (no duplicate intents; no cap violations; deterministic outputs under a fixed seed).

Complexity: High
Suggested files - Add: `ai_trading/replay/event_loop.py`, `ai_trading/execution/simulated_broker.py`, `ai_trading/replay/scenarios.py` - Add tests: `tests/integration/test_replay_async_order_events.py`, `tests/integration/test_parity_invariants.py`

Execution policy selection model and TCA-driven calibration

Why it matters: The repo has TCA logging and execution algorithms, but the report recommends making **execution selection** liquidity/urgency-aware and calibrating expected costs from realized fills (Report page 5). Optimal execution literature frames the cost-risk tradeoff, classically associated with Robert Almgren [8] and Neil Chriss [9]. [10]

Implementation tasks - Implement a `ExecutionPolicySelector` that chooses among: - Passive limit - Marketable limit - TWAP - POV (participation capped) - VWAP proxy (if volume profile available) - Inputs should include: - Spread estimate / quote quality - Recent realized volatility - Liquidity proxy (ADV, recent volume) - Urgency (time-to-close, signal half-life) - Data provenance regime (IEX vs SIP degraded mode). - Implement a lightweight cost model calibration loop: - Compute realized implementation shortfall vs arrival benchmark and store rollups. - Update per-symbol parameters using bounded adaptation (EWMA with clamps) and version parameters so backtests can reproduce. - Ensure report's "arrival benchmark consistency" (decision time vs submit time) is enforced.

Complexity: Medium–High
Suggested files - Add: `ai_trading/execution/policy_selector.py`, `ai_trading/execution/cost_model.py`, `ai_trading/tca/rollups.py` - Modify: `ai_trading/execution/engine.py`, `ai_trading/execution/algorithms.py`, `ai_trading/execution/slippage_log.py`

Model/parameter registry and sensitivity sweeps

Why it matters: The report recommends "small grid sweeps" for stability and a manifest/registry with training/data/embargo/cost-model versions (Report page 5).

Implementation tasks - Add a `ModelManifest` schema (JSON/YAML) and validation: - training window - data sources + entitlements flags (IEX/SIP) - embargo days - feature set hash - cost model version - strategy parameters - Add a minimal registry store in `artifacts/registry/`. - Add a parameter sweep runner that tests perturbations and enforces stability gates.

Complexity: Medium
Suggested files - Add: `ai_trading/registry/manifest.py`, `scripts/run_sensitivity_sweeps.py`, `artifacts/registry/README.md` - Add tests: `tests/unit/test_manifest_validation.py`, `tests/slow/test_parameter_sweep_stability.py`

Alert routing and runbooks

Why it matters: The report distinguishes “metrics exist” vs “operational readiness” (Report page 6). Good SLOs must trigger actionable alerts and automated quarantine/rollback behaviors.

Implementation tasks - Implement notification sinks (email/webhook) and wire them to SLO breach state transitions. - Add runbook docs: “reject spike,” “slippage spike,” “data staleness,” “broker outage,” “kill switch.”

Complexity: Medium
Suggested files - Add: `ai_trading/monitoring/alerts.py`, `docs/runbooks/*.md` - Modify: `ai_trading/monitoring/slo.py` (if present) or equivalent monitor module.

Prioritized task list

Priority	Task	Complexity	Output artifacts
P0	Durable intent/order store + DB-backed idempotency + restart reconciliation	High	DB models + migration, execution-engine integration, integration tests
P0	Replay harness with async order events + parity invariant tests	High	Simulated broker + replay engine + integration tests
P1	Execution policy selector + arrival-benchmark consistency	Medium–High	New selector module + engine wiring + unit tests
P1	TCA rollups + bounded cost-model calibration	Medium–High	Rollup job + versioned cost params + tests
P2	Data provenance regime hardening (IEX vs SIP flags; quote freshness/spread invariants)	Medium	Config + gating logic + tests
P2	Model manifest + registry + sensitivity sweep stability gates	Medium	Manifest schema + sweep script + slow tests
P3	Alert routing + runbooks + auto-quarantine playbooks	Medium	Notifications + docs + small integration tests

Codex Prompt and Delivery Plan

Proposed target architecture

```
flowchart LR
  A[Market Data Ingest] --> B[Eligibility & Data-Quality Gates]
  B --> C[Signal / Edge Estimation]
  C --> D[Portfolio & Risk Targeting]
  D --> E[Intent Store<br/>(Durable)]
  E --> F[Execution Policy Selector]
  F --> G[Order Slicing / Routing]
  G --> H[Broker Adapter<br/>(Live or Simulated)]
  H --> I[Order Events<br/>(Async)]
  I --> J[Reconciliation + Ledger]
  J --> K[TCA Rollups]
  K --> L[Cost Model Calibration]
  L --> F
```

Single Codex prompt

```
You are implementing institutional-grade robustness features described in the
attached report for the repository:

  repo: dmorazzini23/ai-trading-bot
  workdir: repository root

Goal
1) Add a durable OMS intent/order state store with exactly-once semantics.
2) Enforce live/backtest parity by adding a replay/simulation mode that emits
asynchronous order events (fills between bars).
3) Add an execution policy selector and TCA-driven cost calibration so execution
choices reflect spread/volatility/liquidity regimes and improve over time
without destabilizing.

Do NOT change the repo’s public API surface unless necessary; prefer adding new
modules and wiring them into existing engine entrypoints.

Target files to modify (existing)
- ai_trading/execution/engine.py
- ai_trading/execution/idempotency.py
- ai_trading/execution/reconcile.py
- ai_trading/execution/algorithms.py
- ai_trading/execution/slippage_log.py
- ai_trading/database/models.py
- ai_trading/database/connection.py
```

```
- pyproject.toml and/or requirements.txt if SQLAlchemy/postgres drivers/
migrations need explicit pins

New files to add
- ai_trading/oms/intent_store.py
- ai_trading/database/schema.py (optional helper)
- ai_trading/execution/policy_selector.py
- ai_trading/execution/cost_model.py
- ai_trading/tca/rollups.py
- ai_trading/replay/event_loop.py
- ai_trading/execution/simulated_broker.py
- docs/runbooks/restart_reconciliation.md
- docs/runbooks/slippage_spike.md

Data / env configuration
- Introduce DATABASE_URL for Postgres (production) and support SQLITE_URL for
tests.
- Add a config flag SIMULATION_MODE=1 and REPLAY_SEED for deterministic
simulation.
- Add a DATA_PROVENANCE flag (iex|sip|delayed_sip) and expose it to gating +
execution policy selection.

Detailed implementation requirements

A) Durable intent store and exactly-once semantics
- Define an “Intent” object with:
  id (UUID), created_at, symbol, side, target_qty, expected_edge_bps,
strategy_id,
  decision_ts, arrival_price, data_provenance, status (NEW/SUBMITTED/FILLED/
CANCELED/FAILED),
  idempotency_key (unique).
- Implement an IntentStore interface:
  - create_intent(intent) MUST be transactional and enforce unique
idempotency_key
  - get_open_intents()
  - mark_submitted(intent_id, broker_order_id)
  - record_fill(intent_id, fill fields)
  - close_intent(intent_id, final_status)
- Persist intents and order state BEFORE sending any broker order.
- Update idempotency logic: idempotency must be persisted in DB, not in-memory
TTL only.
- On startup (and on each cycle), run reconcile():
  - read open intents from DB
  - query broker open orders/positions
  - deduplicate and correct state (e.g., intent SUBMITTED but broker has no
order => mark FAILED and alert)
  - ensure no duplicate intent causes duplicate broker orders
```

```
B) Simulation/replay parity with async order events
- Implement SimulatedBroker with the same interface the execution engine
expects:
  submit_order(), cancel_order(), list_open_orders(), get_fills()
- In SIMULATION_MODE, order events MUST arrive asynchronously:
  - schedule fills with asyncio tasks that can fire between bar ticks
  - allow partial fills
  - fill price should depend on a simple slippage model (spread + volatility
proxy + participation)
- Add ReplayEventLoop:
  - replays market bars
  - invokes the same decision/risk/execution path as live
  - collects “decision records” and “executed orders”
  - enforces invariants:
    * no duplicate intents for the same decision key
    * no orders exceed caps
    * no trading when data gates block
    * deterministic results with fixed REPLAY_SEED

C) Execution policy selection + cost calibration
- Add ExecutionPolicySelector:
  inputs: symbol, side, qty, spread_bps, realized_vol, liquidity_proxy, urgency,
data_provenance
  output: policy enum {PASSIVE_LIMIT, MARKETABLE_LIMIT, TWAP, POV}
- Implement default decision logic:
  - if data_provenance != "sip" and spread/quote freshness is poor => reduce
aggression (prefer PASSIVE_LIMIT or smaller POV participation)
  - high urgency + tight spread => MARKETABLE_LIMIT
  - large qty / low liquidity => TWAP or POV
- Add CostModel:
  - store per-symbol parameters (temporary impact coefficient, spread penalty,
max_participation)
  - update parameters using bounded EWMA on realized implementation shortfall
  - version the parameters and write them to DB (and optionally a JSON artifact)
so replays are reproducible

Tests (must add)
Unit tests
- tests/unit/test_intent_store_idempotency.py
  * creating same idempotency_key twice returns the original intent and does not
create duplicates
- tests/unit/test_policy_selector.py
  * validate policy outcomes for scenarios (wide spread -> passive, low
liquidity big order -> TWAP/POV)
- tests/unit/test_cost_model_calibration.py
  * EWMA update is bounded; parameters don’t diverge

Integration tests
```

```
- tests/integration/test_restart_reconciliation_exactly_once.py
  * simulate “crash after persist but before submit”; restart should submit once
  * simulate “crash after submit but before persist order id”; restart should
reconcile and not resubmit
- tests/integration/test_replay_async_order_events.py
  * in simulation, fills can arrive between bars; strategy loop still processes
them correctly
- tests/integration/test_parity_invariants.py
  * decision records match executed actions; no invariant violations

Expected behaviors
- Running in live mode without DATABASE_URL should fail fast with a clear error
unless configured to use file-based SQLite explicitly.
- In simulation mode, replay runs are deterministic with fixed seed.
- Idempotency is durable across process restarts.
- Execution policy selection is explainable and logged (policy + features used
to choose).
- TCA rollups produce per-symbol daily metrics including arrival benchmark
slippage and fill ratios.

Deliverables
- Code changes in the specified files
- New modules + tests
- Minimal docs in docs/runbooks/ describing how to recover from restarts and how
to interpret slippage spikes

Make sure all new code is typed, has docstrings, and follows the repo’s existing
style and lint/testing setup.
```

Suggested commit and PR structure

Branch naming - `feat/institutional-grade-parity-and-durability`

Commit sequence - `feat(db): implement SQLAlchemy models and connection layer for OMS state` - `feat(oms): add durable intent store with unique idempotency keys` - `feat(exec): wire intent persistence + exactly-once semantics into execution engine` - `feat(reconcile): restart-safe reconciliation across DB and broker state` - `feat(replay): add simulated broker and async order-event replay loop` - `feat(exec): add execution policy selector and integrate with algorithms` - `feat(tca): add rollups + bounded cost model calibration` - `test: add unit + integration coverage for idempotency, parity, and calibration` - `docs: add runbooks for restart reconciliation and slippage spike response`

PR description template - **Summary:** What report gap(s) this PR closes (durability, parity, execution selection, calibration). - **Design:** Short description of IntentStore schema, exactly-once flow, and replay async event model. - **Tests:** List added tests and how to run them; include deterministic seed behavior. - **Operational impact:** New env vars (DATABASE_URL, SIMULATION_MODE, DATA_PROVENANCE), migration

steps. - **Risk:** Backward compatibility notes; failure modes and safe defaults. - **Follow-ups:** Remaining report recommendations (sensitivity sweeps, model manifest/registry, alert routing).

---

1 3 8 SEC.gov | Risk Management Controls for Brokers or Dealers With Market Access
https://www.sec.gov/rules-regulations/2011/06/risk-management-controls-brokers-or-dealers-market-access?utm_source=chatgpt.com

2 9 13 14 Alpaca Support - Is there a usage limit for the number of API calls per second?
https://alpaca.markets/support/usage-limit-api-calls?utm_source=chatgpt.com

4 10 Journal of Risk Volume 3, Number 2 (Winter 2000)
https://www.risk.net/journal-of-risk/volume-3-number-2-winter-2000?utm_source=chatgpt.com

5 12 Alpaca Support - What data provider does Alpaca use?
https://alpaca.markets/support/data-provider-alpaca?utm_source=chatgpt.com

6 11 R E A D I N G
https://catalogimages.wiley.com/images/db/pdf/9781942471141.excerpt.pdf?utm_source=chatgpt.com

7 Event Handlers - QuantConnect.com
https://www.quantconnect.com/docs/v2/writing-algorithms/key-concepts/event-handlers?utm_source=chatgpt.com