

An evolving population of trading strategies

The Terrarium is a software experiment in adaptive trading. It runs multiple rule-based strategies, gives them capital allocations, evaluates their recorded outcomes, and creates or removes variants over time. Each software organism has a strategy family, adjustable parameters, and a performance history. The organisms are not living systems; evolution is the organizing analogy for selection, mutation, and replacement.

I had considered building a trading bot for some time. Strong technology-company earnings caught my attention around the time I was exploring TimesFM, and I remember sketching The Terrarium on a whiteboard in physics class. That coincidence turned a loose idea into a system I wanted to build. Rather than assume I knew one best strategy, I wanted to explore how a population could respond to a noisy market with sophisticated competitors.

Five starting strategy families

A — Long-term growth. Uses a long moving average, which smooths recent prices, and a momentum condition to seek sustained strength. It exits after persistent weakness relative to the average.

B — Opportunistic trading. Looks for either a rebound after an unusually weak move or a breakout supported by elevated trading volume. Volatility-based loss limits, profit targets, and holding limits constrain the position.

C — Trend following. Combines faster and slower price averages with momentum and trend-strength indicators. When available, a TimesFM forecast can increase or reduce confidence in the proposed direction.

D — Hybrid. Aggregates upstream strategy signals rather than generating an independent price rule. Its design uses accuracy weights and can consult TimesFM when signals disagree; a consensus is still a proposal, not a guaranteed outcome.

E — Fundamental and alternative data. Uses cached company revenue growth, institutional ownership, and insider ownership. It skips a company when the required inputs are missing; ownership percentages are inputs, not proof of informed trading.

Where TimesFM fits

TimesFM is Google Research's pretrained model for forecasting numerical sequences. My wrapper supplies historical closing prices, caches forecasts, and interprets direction for C and D. The inspected code targets TimesFM 2.5 and returns an unavailable, neutral result if loading fails. Integration alone does not establish that a particular recorded run used the model or that its forecasts beat a market baseline.

Sources: [src/evolution/organism.py](#); [src/strategy/](#); Google Research TimesFM documentation, <https://github.com/google-research/timesfm>. The following pages explain selection, Robinhood execution, and recorded results.

Adaptive strategy research platform

I did not know the best trading strategy, and I did not want the system to depend on one remaining useful indefinitely. Nature suggested a different experiment: random variation followed by nonrandom selection. In a noisy market with sophisticated competitors, I wanted to ask how a population might adapt. The Terrarium turns that question into software organisms with strategy parameters, performance histories, changing capital allocations, and replacements for underperforming members.

Contribution

My work includes strategy population management, allocation, risk and execution integration, persistent records, dashboard controls, and replay. The system uses Yahoo Finance market data and a Robinhood broker connection through the robin-stocks library, alongside other open-source Python libraries.

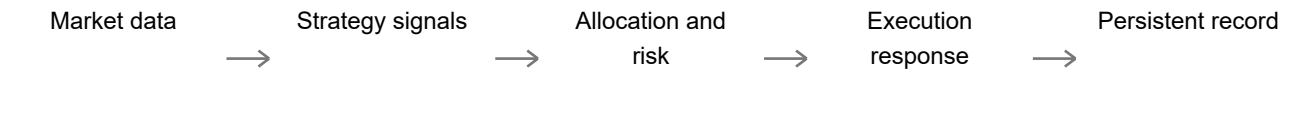


Figure 4. Simplified decision cycle. A proposal, an approved order, and a filled trade are separate states.

Selection and variation

During an evolution cycle, the arena ranks organisms, records extinctions, and requests replacements through mutation or crossover when an eligible leading organism exists. Birth records retain parent IDs and lineage. A diversity check can introduce a new organism; capacity controls remove excess members. These mechanisms explore different parameter combinations, but selection on past outcomes can also reward noise.

Making adaptation inspectable

An evolving population creates a tracing problem: which organism proposed an order, what capital was allocated, and what did the broker actually return? I connected these stages with cycle IDs and committed events in SQLite. The replay interface reconstructs a recorded cycle without submitting another order.

The following pages examine the Robinhood connection, population limits after restart, a rejected sell with no fill, and a saved backtest. They show how the system handles changing state and external responses. The backtest remains a development result, not evidence that evolution produces durable market outperformance.

Implementation references: `src/evolution/arena.py`, `run_evolution_cycle`; `src/data/store.py`, cycle-event persistence.

From strategy signal to broker order

The automatic execution path turns a strategy signal into a sized order, checks it against account and position constraints, and passes approved requests to a broker adapter. Separating these stages lets the strategy propose an action without controlling the external broker directly.

Size and check the proposal

```
246 order = self._signal_to_order(signal, allocation, price, positions)
247 if order is None:
248     continue
249
250 # Risk check
251 risk_result = self._risk.check_order(order, account, positions, price)
```

src/execution/engine.py, lines 246-251. Local source view captured 18 October 2026; HEAD 256c3214.

A rejection either leads to a smaller suggested quantity and a second check, or skips the order. Broker-capacity limits and stored intent checks follow before submission. This separates a strategy's proposed action from what the account can execute and suppresses repeated unchanged requests.

Submit through the Robinhood integration

```
245 if is_fractional and order.notional is not None:
246     func = lambda: r.orders.order_buy_fractional_by_price(
247         symbol=symbol, amountInDollars=round(order.notional, 2), timeInForce="gfd"
248     )
249 elif is_fractional:
250     func = lambda: r.orders.order_buy_fractional_by_quantity(
251         symbol=symbol, quantity=qty, timeInForce="gfd"
252     )
```

src/execution/broker.py, lines 245-252. Local source view captured 18 October 2026; HEAD 256c3214.

```
275 try:
276     res = await loop.run_in_executor(None, func)
```

src/execution/broker.py, lines 275-276. Local source view captured 18 October 2026; HEAD 256c3214.

Selected fractional-buy branches in RobinhoodBroker; other buy and sell branches are omitted. The engine calls this adapter through `await self._broker.submit_order(order)`.

Submission is not execution

The adapter reads the returned order ID, status, cumulative filled quantity, and average price, then polls for an updated order status. The engine records the result and updates its stored intent. The 17 October startup log confirms selection of RobinhoodBroker. Submission can still fail or produce no fill, as the CRM replay shows; this connection is evidence of implementation, not profitability.

Persistent events and restart handling

The cycle store writes a JSON payload with each event and commits it to SQLite. Retrieval orders events by database ID. That preserves a concrete sequence for inspection, including events written before a later operation fails.

```

706     async def log_cycle_event(self, cycle_id: str, event_type: str, payload: dict[str, Any]) ->
      None:
707         """Append one immutable, JSON-serializable event to a trading cycle."""
708         assert self._db is not None
709         await self._db.execute(
710             "INSERT INTO cycle_events (cycle_id, event_type, payload, created_at) VALUES (?, ?,
      ?, ?)",
711             (cycle_id, event_type, json.dumps(payload, default=str),
      datetime.utcnow().isoformat()),
712             )
713         await self._db.commit()
714
715     async def get_cycle_events(self, cycle_id: str, limit: int = 500) -> list[dict[str, Any]]:
716         assert self._db is not None
717         cursor = await self._db.execute(
718             "SELECT * FROM cycle_events WHERE cycle_id = ? ORDER BY id ASC LIMIT ?", (cycle_id,
      limit)
719         )
720         rows = await cursor.fetchall()
721         columns = [d[0] for d in cursor.description]
722         return [{"**dict(zip(columns, row)), "payload": json.loads(row[3])} for row in rows]

```

Source screenshot of src/data/store.py, lines 706-722. Omitted lines are marked.

Listing 3. Current cycle-event storage and retrieval. An immediate commit preserves recorded events; it does not independently verify the data or broker response contained in them.

Restart invariant

A second persistence problem concerned population size. Loading organisms from storage could restore more members than the configured arena allowed. The startup path now passes the restored population through arena seeding and records culled organisms with the cause `startup_population_cap`.

```

233
234 # Seed the arena & handle excess culled organisms
235 culled = self._arena.seed_population(population)
236 for org in culled:
237     await self._terrarium.record_extinction(org, cause="startup_population_cap",
      fitness=0.0)

```

src/orchestrator.py, lines 233-237. Working copy inspected 15 October 2026; HEAD 256c3214.

Listing 4. Startup culls are written back to the lifecycle record. Capacity is applied when reconstructing runtime state rather than assumed to hold because the state was saved.

Recorded CRM order rejection

Cycle 353 on 28 September 2026 provides an observed failure case. Its events connect a CRM sell proposal to allocation and an external rejection. The recorded fill is zero.

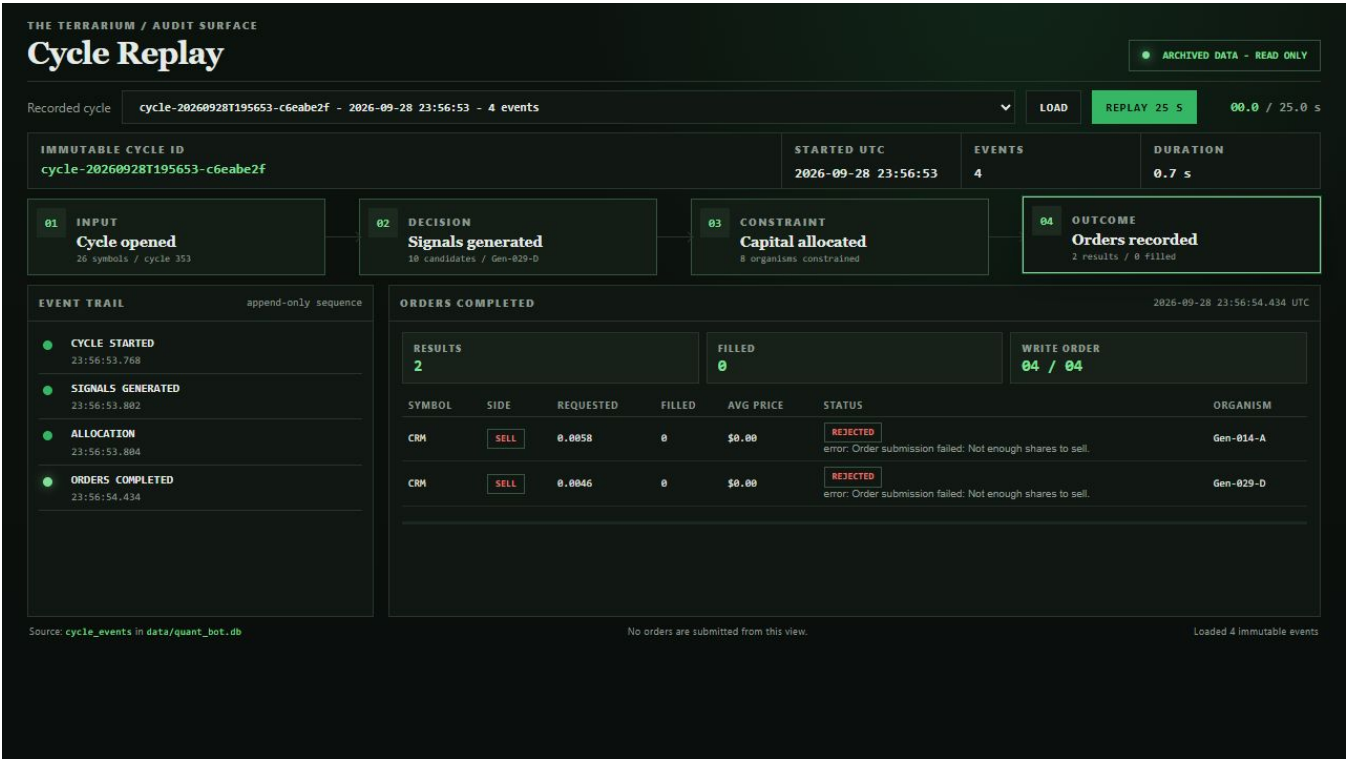


Figure 5. Working replay interface captured on 18 October 2026 through a local read-only database viewer. The displayed events were recorded on 28 September; this is a replay, not a new trading run.

Observed outcome

Gen-014-A requested 0.0058 CRM shares and Gen-029-D requested 0.0046. Both responses reported “Order submission failed: Not enough shares to sell,” with zero shares filled. A query matching the Gen-029-D order ID and submission timestamp finds no trade row.

Interpretation

This cycle demonstrates that the event chain retained a rejected request rather than reporting a completed sale. It also exposes a holdings-state issue. The record alone does not explain or resolve that discrepancy, and it is not evidence that every rejection is automatically reconciled.

Source: cycle_events rows 44752–44755 and matching-order query against the trades table.

Recorded backtest and limitations

The dashboard backtest workflow executes the repository backtester, persists the request and result, and permits later retrieval. The run below is a historical pipeline result from 25 September 2026.

Run parameter or result	Recorded value
Requested evaluation window	27 June to 25 September 2026
Latest loaded daily bar	24 September 2026
Partition	C Indicators and Trend
Symbols loaded	26
Initial simulated capital	\$100
Net return	+0.89%
Maximum drawdown	1.70%
Closed trades	21

Table 2. Persisted experiment dashboard-backtest-3372353706b6. Values are rounded from the stored result, not recomputed performance estimates.

Method and interpretation

The documented engine used daily historical bars and 0.10 percent simulated slippage at entry and exit. Signals used a bar's close and executed at that same close. The run did not include a matched benchmark, commissions, spread, market impact, or a frozen data snapshot. These limitations prevent interpreting the result as evidence of alpha.

The stored base revision bdfdd36b does not include the then-uncommitted dashboard implementation. Consequently, the revision identifier alone cannot reproduce the full run. A credible strategy comparison requires a frozen implementation and data, causal execution timing, matched costs, and an untouched evaluation period.

Source: experiments table, run ID above; configuration hash 6e66b029de5c4593.