Prospector: an unattended arbitrage system, and the failures I found in it

Prospector is a private system that looks for retail arbitrage opportunities for Amazon FBA. It runs on its own every night on a VPS, pulls data from five external services, evaluates each product against a configurable set of business criteria and sends an alert when it finds something worth buying. It has been running since 21 July 2026.

It is a single-user system. The user is me. The backend is Python (15,315 lines of production code), the dashboard is TypeScript, and Supabase is the contract between them. I do not write the code by hand: I direct Claude Code. What I contribute is deciding what gets built, noticing when something is wrong even though every indicator is green, and verifying results by running them.

Figures as of 12 August 2026: 159 commits, 73 numbered decisions each recording the incident, the cause and the verification, 806 test functions in the backend across 66 files, 240 in the dashboard, and 46 database migrations. Nineteen overnight runs of roughly four hours each, 16,803 product evaluations, and four production failures with documented root cause.

The system is built to authorise real purchases and it spends real money on paid APIs. In three weeks it has not once said "buy". This document explains why that zero is the design working as intended, and what broke along the way.

Inventory of failure modes

Failure	How it appeared	How it was found	Response
Out of memory on the nightly process	3 runs ending in `exit 137`; the wrapper recorded the exit code, the application recorded nothing	Kernel `dmesg`, after noticing two days with no candidates	Trimmed the retained payload, trimmed the parsed series, added 4 GB of swap
Sustained Amazon rate limiting	1 run ending in `exit 1`; 3 h 37 min of work lost	Run log traced to a line with no `try/except`	Rate pacing, per-product tolerated failure, circuit breaker
Silent degradation	A run reporting success with zero useful results	Adversarial review of the previous fix	Breaker after 8 consecutive deferrals, alert if 50% or more defer
PostgREST 1,000-row cut	Incorrect figures, not truncated ones, on 4 screens	Comparison against the database	Dedup, filter and anti-join moved into the database; deterministic pagination
Valid but impossible data from the Amazon API	`Status=Success` with 201.77 USD of shipping on a 25.95 USD sale	Manual review of the first real run	Physical sanity check: dimensional weight against actual weight, then human review
Half-cent rounding	Profitable products classified as doubtful	Sweep of 2,004 combinations	Re-verification using the already-rounded cost
Test suite not hermetic	Everything green, but the tests would have called paid APIs	Review found the factories select the real adapter when credentials exist	Fixture that clears 9 credentials before every test
Seller harvesting broken (**open, unresolved**)	HTTP 400 from Supabase; the run still ends in `exit 0`	Logs of the three most recent runs	Not fixed, no alert, undocumented until now

The memory failure is worth describing in full because it is the most instructive. Every product retained the raw Keepa payload, about 645 KB, of which 96% was historical price series. The run accumulates all products in memory before filtering, which reached roughly 3.3 GB. That data was write-only: it was persisted for traceability and nothing ever read it back. The fix had three layers. Dropping the heavy keys took each product from 645 KB to 22 KB. Trimming the parsed series to the window the filters actually use took them from 3,516 points to 218 per product. A 4 GB swap file was added as a safety net. The run now uses between 300 and 500 MB.

The part that mattered most is how it failed. `SIGKILL` does not let Python run its `try/except`, so the application never recorded a failed run. The system spent two nights in complete silence, which is exactly what it reports on a night with no opportunities. For an unattended system, the expensive failure is not crashing. It is crashing in a way that looks identical to working.

Design decisions and the alternatives I rejected

Decision	Alternative	Reason
24 `Protocol` contracts with real and test adapters selected by configuration	Calling the APIs directly from the business logic	Testing something that handles money cannot cost money. The suite runs offline in seconds
An `autouse` fixture that clears 9 credentials	Trusting that the test machine has none	The VPS is also the development machine
Fail closed when parsing store data	Guessing the format of a new store	Misreading a price means buying at the wrong price
Fail open on the manual veto list	A uniform error policy	The worst case is an unwanted product reappearing; the opposite blinds the whole catalogue
Defer the product when the fees API fails	Estimating the fee	An honest gap is more useful than an invented number
Adaptive API budget	Spending and seeing what happens	The quota is use-it-or-lose-it and does not accumulate

The error policy is deliberately not uniform. It is chosen according to the economic consequence of each failure and written down next to the code. The finest distinction is in the fees API: a 429 on one product defers that product to amber and the run continues, while an authentication error is deliberately excluded from the tolerated-failure list and stops the run. A broken credential is a global problem, and reporting "missing data" on two hundred products would be misleading. The run also does not declare a global failure when no seller was attempted, because that case is the budget brake working correctly.

The circuit breaker came out of a review rather than the original design. The first version of the rate-limit fix only handled per-product failures. If Amazon had gone down entirely, every product would have come back without data and the run would have completed successfully with zero results. These are two different defences: the breaker opens after 8 consecutive deferrals and stops hitting an API that is down, and a Telegram alert fires when 50% or more of the products that attempted a fee lookup were deferred.

Integration: five APIs that fail in different ways

There are no inbound webhooks. Everything is polled from cron, using my own `httpx` clients rather than vendor SDKs, with `tenacity` backoff, secret redaction in logs and structured JSON output. Keepa and the Amazon Selling Partner API each have an independent token bucket. Before building the fees client I checked the official changelog and confirmed that the Product Fees API uses LWA authentication without AWS request signing. The SerpAPI parsing profiles are data: each store declares how its response differs, and an engine with no profile is skipped with a warning rather than guessed at. Rate pacing runs at half the documented limit, and `retry-after` is honoured separately by the client backoff, which is a second layer.

All arithmetic uses `Decimal`. Criteria return three states: pass, fail and not evaluable. The count only includes evaluable criteria, so missing data is never counted as a pass. When the store tax cannot be determined unambiguously, the most conservative value is applied, because guessing low inflates the return on investment.

The subtlest money defect was a half cent. The formula for the maximum purchase price was correct, but the cost actually paid is rounded again to whole cents, and that left the result just below the target in about 7% of cases where tax

applied. The fix does not touch the formula. It re-verifies using the already-rounded cost and lowers the ceiling one cent at a time. It only ever lowers it.

The PostgREST row limit was the other one. The dashboard was deduplicating and counting in JavaScript over whatever it received, and because the ordering was by UUID, the 1,000 rows it received out of 2,827 were effectively random. That meant financial projections calculated over a random sample. One view displayed 154 items when the real figure was 422, and an anti-join over 8,509 truncated rows could inflate the committed capital. After closing the first occurrence, the useful question was where else the same pattern lived. It lived in three more places. The last one was still latent and was validated with synthetic parity testing, 60 cases out of 60 with no discrepancies.

The zero, without dressing it up

Across the 15 complete runs the system performed 16,803 evaluations over 3,444 distinct products, since the same product is re-evaluated each night. The green verdict came up zero times. The most recent run: 1,061 evaluations, 0 green, 488 amber, 573 red.

The filters being strict is only part of the explanation. Of those 3,444 products, only 16 have a calculable profit, because the rest come from storefronts with no verifiable purchase price. Without a source cost there is no return figure, and without a return figure the system will not mark anything green. It leaves the product amber or sends it to the watchlist. The bottleneck is the quality of the sources rather than the filter, and that is where the architecture went next. When real money is involved, a false positive costs money and a false negative costs waiting.

The morning alert only contains green results. When there are none, nothing is sent and the silence is recorded in the log, which has happened 15 times. A daily "nothing today" alert trains the reader to ignore alerts.

What the system does not have

Three weeks of recorded operation rather than months: 19 runs across 22 days, with cron deliberately paused from 24 to 27 July while the memory fix was completed. There is no continuous integration; tests, `ruff` and `mypy --strict` are run by hand before each merge. There is no coverage measurement. The test fixtures are static snapshots, so if Keepa changes its schema I will find out in production. The circuit breaker exists only in the fees subsystem. The health check has 7 verifications and has been executed exactly once, manually, because it is not in cron. The money calculator is duplicated in Python and TypeScript and is held together by parity tests. And of the 5,551 recorded cron executions, 5,336 are a poller that runs every three minutes and usually has nothing to do, which is correct behaviour but does not scale.

What I take from it

The three most expensive failures produced no error. The memory kill left no trace in the application, the row limit returned incorrect data with `200 OK`, and Amazon correctly returned shipping dimensions for a box the size of a dishwasher. Reading error logs is not where the work is. The work is in not trusting the green.

When a defect is found, the fix rarely ends where the defect was visible. The same pattern tends to appear somewhere nobody looked. While writing this document I confirmed that seller harvesting has been failing for three consecutive nights with a 400 from Supabase, tolerated by a `try/except` that logs the error and returns zero results. That is the exact problem this document criticises, happening in my own system, today. It is unresolved, and I would rather write it down than have someone else find it.

The uncomfortable part of closing a portfolio piece this way: the system has not yet produced a single buy signal. What has been demonstrated is that it runs unattended, filters, stops itself when the budget runs out and reports when it degrades. Not that it finds profitable opportunities. I could change that in five minutes by lowering a threshold, and it would be the worst decision in the project.

Raymond Reyes · linkedin.com/in/raymondreyesh · raymondreyesh@gmail.com