The Factory: Brownfield Use Case

An autonomous engineering team operating inside a live production system.

The below is an example of BLA's Factory working in a production system that it didn't build, with live users, with legacy data, and no tolerance for breakage.

The client has a production financial-analytics platform that serves portfolio values, returns, and holdings to clients over an API. Underneath it are three data stores with three different consistency models (a time-series store, a relational store for the ownership hierarchy, and a distributed-lock store), an event-driven pipeline that rolls positions up a multi-level hierarchy (positions to accounts to owners and trusts to top-level entities), and a legacy migration in its past that left subtle, client-visible drift between the analytics store and the upstream source of truth. That drift is the origin of most incidents.

We applied the Factory to help clean up those subtle issues and remove any data inconsistencies that harm platform trustworthiness.

What makes it trustworthy: Council Mode

Any tool can produce a confident answer. The one that hurts in production is the confidently wrong one. Before any load-bearing decision the Factory runs Council Mode: a position is put forward, a critic from a different AI provider family attacks it, the original agent rebuts, and the result is re-evaluated before a human gate. The governing rule is that no single agent can lock in a load-bearing decision. Different model families fail in different places, so this is a second prior, not a second opinion. High-stakes calls go to a panel. Where a script can settle the question it writes the script. When the models disagree, that disagreement goes to a human rather than being averaged away. That is why an autonomous agent can be allowed to write numbers a client sees.

What it did

A client bug, root-caused and proven. A senior stakeholder flagged one holding: its value looked right, but its returns were nonsense, a price return of -100% and a deeply negative time-weighted return on an asset that was actually up. The Factory pulled the live served numbers and confirmed the paradox, then read how returns are computed and found a non-obvious coupling: returns come from a different rollup than values. Its first check looked in the wrong store; it caught that, said so, and

rechecked in the correct one before making any claim. It proved the cause with a scoped experiment, rebuilding that one asset's return data and watching -100% correct to the right +38%, now consistent with the value-based return. Then it applied the same rebuild across every sub-entity of the portfolio, serialized so it wouldn't overload the shared pipeline, and reported what was fixed, what caused it, and what was still off: a couple of legacy-migration placeholder assets and a separate category-level issue. The lasting lesson it wrote down for the team: re-rolling rebuilds values but not the return sub-periods, so those have to be re-run afterward.

A 190,000-point reconciliation, reversible and verified. A large portfolio's history had drifted from the source of truth after the migration. The Factory snapshotted roughly 190,000 data points across the full hierarchy first, so the operation was reversible. Before writing anything it verified that the underlying positions already matched the source of truth, so a rebuild would converge rather than bake in corruption; had they not matched, it would have stopped and reported instead. It worked to a written plan: split the history into windows, rebuild one slice, verify it to the cent, then commit the rest. When a faster heuristic looked tempting, a panel of different models attacked it and showed it would silently under-count a subset of cases, so it kept the slower method it could prove. Every level (positions, accounts, owners, trusts, and the top-level total) matched exactly; the one residual was a known upstream migration artifact, flagged rather than hidden.

The outage that wasn't. An alarm reported thousands of failed jobs on the second day running, and the on-call engineer was about to mass-retry. The Factory pulled the arrival-rate metrics and showed it was not escalating: about 94% of the backlog was stale fallout from a bug that had already been fixed, and there was no data loss, because the failures were captured by design rather than dropped. It reprocessed only the genuinely actionable items, refused the blind retry that would have re-corrupted client data, scoped the retry to the entities it had verified were safe, and opened a PR for the one real remaining bug.

The fix, shipped under the team's rules. That bug was a long-running job timing out on very large inputs. The Factory wrote the fix as a chunked job that does a bounded amount of work per run and resumes itself, added a unit test for the correctness property, and passed the project's lint bar. It built against a clean copy of the deployed code, which surfaced a latent signature mismatch that would have crashed production, and opened a reviewable PR following the team's branch and deploy conventions.

The bottom line

The Factory diagnoses from real data, traces a wrong result to the store, rollup, or line that produced it, and puts every load-bearing conclusion through Council Mode. It snapshots before writes, scopes the blast radius, and never hand-deploys; for regulated or sensitive data it runs on self-hosted models on our own hardware, so nothing leaves the machine. The hard part of production is not writing code; it is understanding a system you didn't build, changing it safely, and proving you didn't break it. That is what the Factory does.