The Factory and Siegfried

Two software problems, one cause

Most organizations face software from two directions at once, and both directions are blocked by the same thing.

The first is building new software. Every platform reaches a point where its backlog is the most accurate description of its limitations: every unresolved bug, every deferred feature, every piece of technical debt that survived the last three planning cycles. Teams learn to manage backlogs, but they rarely clear them, because issues are generated faster than they are resolved.

The second is the legacy estate. European enterprises spend roughly 55 billion euros a year maintaining software more than a decade old, and the EU Commission has found that around 7 in 10 European enterprises run core software more than 10 years old. That has been a rational decision for years. The system works, the team knows how to keep it running, and a replacement is expensive and risky. The trap is that the cost of staying still rises every year while the cost of moving falls.

These look like different problems. They have the same cause. In both cases, the organization cannot produce or renew software at the speed and safety the situation now demands, and adding engineers does not fix it, because the constraint is not headcount. It is the cost of doing the work correctly: understanding the system, reviewing the change, integrating it, and proving it is safe. Qaori addresses both with one capability: a governed autonomous factory.

Why now

AI-assisted coding changed part of the equation and exposed the rest. Code generation became fast. What generation tools did not make faster is everything around generation: the review that catches what generation gets wrong, the integration that confirms a change did not destabilize the system, and the governance that keeps the result correct, secure, and consistent. You can now generate code

Qaori.Tech OÜ, Sepapaya 6, 15551 Tallinn, Estonia, Register 17169588.
Qaori UK LIMITED, 66 Paul Street, London EC2A 4NE, UK, Register 16952497.
Qaori AI LLC, 1207 Delaware Ave #1634, Wilmington, DE 19806, United States, Register 61-2310301
Qaori AI Japan, Minato Mirai, Yokohama 220-0012, Japan. (Launch Q3 2026)
Qaori AI Inc., 1207 Delaware Ave #1634, Wilmington, DE 19806, United States, (Registration ongoing)
Qaori AI Tecnologia Brasil Ltda., Rio de Janeiro. (Launch Q4 2026)
Qaori.Coop is a service operated by Qaori Tech OÜ, Estonia, registered as Qaori DAO LLC registered at 108 West 13th Street, Wilmington, New Castle, Delaware, 19801, USA.

© 2026 Qaori.Tech OÜ. All rights reserved. Patents Pending - USPTO Provisional Application No. 64/084,387

faster than you can safely review, integrate, and govern it. The bottleneck survived the first wave of AI tooling, and it moved downstream.

At the same time, modern-stack competitors are compounding velocity. A small team building greenfield with modern tooling can ship what once took a far larger team years. A legacy-constrained organization cannot close that gap by working harder, because its core systems do not support rapid iteration. Meanwhile, the engineers who built those core systems in the 1980s and 1990s are retiring, and the business logic that exists nowhere except in their heads and in the code is leaving with them. Both pressures point to the same need: a way to build and renew software that is fast and governed simultaneously.

The principle: a governed factory, not a code generator

The Qaori factory is not a code generator. It is a governed pipeline. The difference decides whether an AI development system is a productivity tool or a production capability.

A code generator accelerates one engineer's work. A governed factory replaces the unstructured, variable-quality process of manual development with a structured, policy-enforced process that is AI-executed. It is not faster development that skips governance. Governance-enforced development is faster than ungoverned manual development. The design position underneath it is simple: automation does the work, automation does well, consistent execution, complete coverage, and pattern matching at scale, and humans make the decisions that require judgment.

How the factory works

The factory carries each unit of work from intake to deployment through a sequence of defined stages, each with defined inputs, outputs, and failure conditions. The buyer does not need the stage list. The governance is what matters, and it rests on a few non-negotiable commitments.

An intake gate that enforces requirements quality. Before any work enters the pipeline, QAPPA evaluates whether the specification is complete, the acceptance criteria are explicit, the dependencies are resolved, and the scope is bounded. Work that fails is returned with a structured report of exactly what was missing. There is no bypass. The factory does not start work it should have rejected, which is the failure mode that destroys ungoverned automation. QAPPA is the most consequential gate in the factory for the reason explained in the next section.

A runtime policy supervisor. Every action the factory considers, every file change, every external call, every environment modification, is evaluated against policy before it executes. Actions outside the permitted scope are blocked. This is enforcement at

the level at which actions are taken, not a check that a cleverly constructed prompt can talk its way around.

Two mandatory human gates. The pipeline cannot reach production without a human review of the implementation and a separate human decision to merge it. These are architectural requirements, not workflow conveniences. There is no configuration that removes them and no automation that substitutes for them. Each produces a signed, timestamped record in the audit trail. The factory generates, tests, and proposes. Humans review and merge.

Automated quality, security, and recovery. Tests, security scanning, and a set of non-negotiable coding standards are enforced in the build infrastructure rather than left to reviewer discretion, so insufficient coverage or a standards violation is a hard failure, not a finding to argue about later. When something breaks, a recovery layer detects the failure, attempts a defined recovery, and escalates to a human with a structured report when it cannot.

This architecture was not theoretical from day one. An early, under-governed version of the factory ran without an intake gate, a recovery layer, or a runtime supervisor, damaging its own working environment. The response was not to abandon autonomy but to add exactly those three controls. We document that openly because a system's credibility comes from how it responds to the conditions it was not designed for, not from a claim that it was always perfect.

QAPPA: defects are born in the requirements, not the code

The reason QAPPA is the most consequential gate is one of the oldest and best-established findings in software engineering. The majority of software defects, commonly estimated at around 70%, do not originate in coding. They originate in the requirements: ambiguous specifications, missing acceptance criteria, unbounded scope, and assumptions that were never made explicit. A defect introduced in the requirements and caught only in production costs many times more to fix than it would have at the definition stage. Code generation, human or AI, cannot fix a flawed requirement. It can only build it faithfully and at speed, which means that a faster code generator, when pointed at weak requirements, simply produces defects faster.

This is why the gate that matters most sits before the first line of code, not after it. Most AI development tools accelerate generation while leaving the requirements untouched, thereby accelerating the wrong thing. QAPPA does the opposite. It refuses to let work enter the factory until the requirement is sound, which is the single highest-leverage point for preventing defects.

QAPPA is, in effect, an autonomous Product Owner, and a deliberately uncompromising one. It is an Alter in the full sense described in the Wisdom Engine

and Alters paper: a governed persona with a defined mandate, here the mandate of holding every incoming requirement to the standard a great Product Owner would. The discipline it enforces is the discipline a great Product Owner applies to every story: a clear specification, explicit and testable acceptance criteria, resolved dependencies, and a scope small enough to reason about. The difference is that a human Product Owner applies that discipline unevenly, under deadline pressure, across a backlog larger than any one person can hold to a consistent standard, while QAPPA applies it identically to every work package, every time, with no fatigue and no pressure to wave something through. The craft encoded in QAPPA comes directly from three decades of training Product Owners; the aim was to capture the standard that the best of them hold to and make it the floor for every item the factory builds. Requirements quality stops being the part of the process most often skipped and becomes the part that is enforced first.

QAPPA belongs at the entrance, and not only the factory's. The same gate applies wherever a requirement meets execution: a feature request, a change to a governed system, a brief handed to an Alter, a scope proposed to Siegfried for transformation. Anywhere weak requirements would otherwise pass downstream and become expensive defects, QAPPA is the discipline that stops them at the door. It is best understood as a reusable platform primitive for requirements quality, with the factory as its first and most demanding application.

Seeing the factory work

A factory you cannot observe is a factory you cannot trust. The Glass Factory is the live view that makes the factory's operation visible in real time: which work package is in flight, where it is in the pipeline, every governance gate decision as it happens, the running governance score, and a record of every action the policy supervisor evaluated and what it permitted. An engineering leader can watch not only what the factory did, but what it was evaluated as permitted to do before it did it. A scripted demo proves you can animate a screen. A live factory running against real infrastructure proves you built what you claimed.

Building new software

The same factory that builds the Qaori platform itself is available as a product that runs in the client's environment, against the client's codebase, in the client's cloud account, fully isolated, governed by the same pipeline and the same gates. No Qaori infrastructure is shared, and one client's work and audit trail cannot be reached from another's.

The primary use is clearing a backlog that manual development cannot keep pace with. The factory processes work systematically at a governed quality, with human review

and merge gates applied to every item. Engineering attention shifts from execution to decision, which is a far more leveraged use of an engineer's time. A small team can review and merge the output of a factory processing many times the issues they could close by hand. It is a force multiplier for engineers, not a substitute for them, and the mandatory human gates are not a feature that can be disabled for a fee.

The outcomes are what make the case: the large majority of well-specified work packages are completed without human intervention beyond the two gates, at a cost per issue measured in single dollars, in minutes rather than hours for simple work, and at a throughput of tens of issues a day. These are the result of making oversight automatic and consistent rather than removing it.

Transforming legacy software

The legacy estate is not a technical problem. It is a strategic one and the second face of the same capability.

The argument for deferral always includes the same line: we cannot replace this system because nobody fully understands it. That is almost always true, and it is almost always offered as a reason to wait when it is the strongest reason to act. Legacy systems encode business logic that exists nowhere else: how an order is validated, how a regulatory case is handled, how an edge case that appeared once in 1994 is managed. It is in the code, rarely in documentation, and increasingly only in the memory of people about to retire.

The traditional fix is an agency-led rebuild: interview the remaining engineers, write documentation, specify a replacement, and build. The model has a well-documented failure rate because human analysis of a system that has accumulated thirty years of patches cannot go deep enough within an engagement budget. The specification diverges from reality, the rebuilt system handles the documented cases and mishandles the undocumented ones, and the undocumented ones are often the ones that matter most.

Siegfried takes a different approach. It reads the legacy codebase directly and exhaustively, across the languages these estates are actually built in, including COBOL, RPG, Delphi, and older Java and Basic dialects, and produces a complete model of the system's dependency graph, business logic, and technical debt. That model is not a summary for human interpretation. It is a structured specification that feeds the same governed factory that builds new software. This changes the economics in three ways. It removes the multi-month discovery phase because the analysis is the first deliverable. It closes the specification-to-reality gap because the specification is generated from the code rather than from memory, including edge cases. And it replaces variable agency build quality with consistent, governed, auditable factory

quality. Transformations that previously took twelve to twenty-four months can be completed in weeks for well-scoped systems, not because corners are cut, but because the bottleneck that slowed rebuilds has been automated away.

The isolation principle

One design decision makes the rest possible, and it is the clearest expression of why this is a Qaori capability rather than a generic one: the source code never leaves the client's environment.

Siegfried is deployed into a cloud account that the client provisions and controls. The analysis, the specification, and the build all happen inside that account. The Qaori control plane receives governance evidence, decision logs, approval records, and audit trails, and never receives the source code. When a client hands a legacy rebuild to a traditional agency, they hand over the code that encodes their business logic and their competitive differentiation. Siegfried does not require that. The transformation happens on infrastructure the client controls, the rebuilt system belongs to the client, and when the engagement ends, the tooling is removed. This is a sovereignty decision, and it is architectural rather than contractual. We could have built it the other way. We chose not to because the alternative would ask the client to trust us with something they should not have to.

After a transformation, the same factory maintains the renewed system. The standards established during the Siegfried engagement become the standards the factory enforces, so the system rebuilt with discipline remains disciplined. Build, transform, and maintain are one continuous capability rather than three procurements.

Governance and evidence

Both halves are governed the same way as the rest of the platform. Every unit of work produces a complete audit record from intake to merge: the specification, the triage result, every stage transition with its timing, the policy supervisor's decisions, the human review, and merge records with identities and timestamps, and the test and security results. That record is the artifact through which an autonomous build system demonstrates compliance, and it maps directly onto the EU AI Act's logging obligations for automated systems. The audit trail is not generated for compliance. It is generated because a system with this level of autonomy requires complete auditability, and the compliance alignment follows from the design. For clients running Sentinel alongside the factory, every factory action also passes through the governance runtime and contributes to the same pre-execution evidence chain described in the Sentinel and Preemptive Governance papers.

What changes for the buyer

For a **CTO or engineering leader**, the backlog stops being a permanent description of what you cannot get to. The factory ensures it meets the governed quality, and your engineers spend their time on architecture and decisions that require their judgment, rather than on routine execution.

For a **CIO facing a legacy estate**, the choice is no longer between an expensive, risky agency rebuild and continued deferral while the people who understand the system retire. The system is read exhaustively, specified against reality, rebuilt in weeks rather than years, and the source code never leaves your environment.

For an **investor**, this is a distinct USP that few can claim together: build new software and transform legacy software on one governed, auditable, sovereign factory. It is also a natural land-and-expand motion. A backlog engagement or a legacy transformation opens the door, and the same factory maintains what it built under the same governance enforced by the rest of the platform.

Conclusion

Organizations are blocked on software from both ends: they cannot build new systems fast enough, and they cannot safely renew the old ones. Both are the same constraint, the cost of doing the work correctly, and both yield to the same answer: a governed autonomous factory where automation executes and humans decide, the work is fully audited, and the client's code stays on the client's soil. Build new, transform old, maintain both. One factory, governed by design.