CONTENTS

THINKING EXPERIMENTAL

The Orchestra and the Codebase

Marcus Persson Rydberg · March 2026 · ↓ PDF

Professional orchestras have solved a problem that software teams keep failing to solve: how to produce reliable, high-quality output at scale, from a large group of specialists, under pressure, repeatedly. The solution isn't talent. It's the rehearsal structure.

The Rehearsal Problem

Before a professional orchestra performs, they don't just gather the musicians and play the piece. They work through a structure that looks almost compulsive in its thoroughness: individual practice, then section rehearsals, then combined sections, then full orchestra, then dress rehearsal in the performance hall. Each stage is required. None can be skipped without specific consequences.

The structure is not about thoroughness as a value. It's about physics. Each stage introduces a new context — more interacting parts, different environmental conditions

— and each new context reveals errors that were literally invisible at every prior stage. Not harder to find. Absent. A violinist's intonation against the second violin cannot be heard in solo practice because the second violin is not there. A balance problem between the woodwinds and the full orchestra cannot surface in woodwind section rehearsal because the orchestra hasn't played yet. A stamina failure in the third hour of a demanding performance cannot be caught in any rehearsal that isn't three hours long under performance conditions.

The rehearsal hierarchy is the mechanism by which a complex system — many specialists, many interacting parts, high stakes if it fails — is made predictable. Skipping any ring doesn't mean fewer rehearsals. It means those errors go undetected until a later ring, where they are more expensive to find and fix — or to the performance, where the audience is watching.

Software engineering has the same structure. It has largely the same solution. It is much worse, on average, at applying it.

The Hierarchy

The rehearsal rings build on each other. Each one is a necessary condition for its class of errors to become detectable.

INDIVIDUAL PRACTICE

Each musician learns their part alone. The errors caught here are entirely local to the individual: wrong notes, technical failures, misread phrasing. What cannot be caught: anything relational. Intonation against another musician, ensemble timing, balance — these phenomena don't exist in solo practice. The ensemble must be present for them to manifest.

SECTION REHEARSAL

The string section rehearses together; brass, woodwinds, percussion separately. New errors become visible: two violinists with individually correct phrasing whose interpretations clash against each other. Intra-section timing drift. Balance within the section. None of these errors had any existence at the individual practice stage — they require multiple musicians playing simultaneously to appear.

FULL ORCHESTRA REHEARSAL

The first time all sections play together. The conductor hears things that were impossible to hear in any section rehearsal: the woodwind melody that sounds fine in isolation but disappears into the full texture, cross-section timing that each section had individually correct but together is slightly off. A cello section's perfect legato becomes inaudible in the final movement because of dynamic imbalance with the brass — a problem that has no existence until the full ensemble plays.

DRESS REHEARSAL

Full performance conditions: the performance hall, the lighting, no stops. The class of errors only the dress rehearsal can catch: stamina (can the orchestra sustain the piece?), the acoustic difference between an empty hall and a full one, and the psychological variables of performance conditions — the specific way the nervous system responds to a thousand people listening. These errors cannot appear in any previous ring because those conditions are absent.

The dress rehearsal is non-negotiable not as a matter of professional culture but as a matter of structural necessity. It is the only ring that tests performance-condition errors. Those errors have no other detection path. Skipping it means the performance is the first time performance conditions are tested — and the audience is watching.

The Software Parallel

The mapping is not a loose analogy. It is structurally precise.

ORCHESTRA	SOFTWARE	ERROR CLASS FIRST VISIBLE HERE
Individual practice	Unit tests, linting, type checking	
Section rehearsal	Integration tests with real dependencies	
Full orchestra	System tests, staging environment, load testing	
Dress rehearsal	Pre-production run, canary deployment	
Performance	Production	

The structural argument holds: a unit test cannot detect a database deadlock that requires three services writing simultaneously under load. That condition is absent at the unit test level — not harder to detect, absent. Similarly, staging catches configuration drift that system tests cannot catch, because system tests run against the system test environment, not production infrastructure. Only production conditions can test production conditions.

This is why the rings cannot be compressed. More unit tests cannot substitute for integration tests. More integration tests cannot substitute for system tests. More staging validation cannot substitute for a pre-production run. Each ring is a necessary condition for its error class to be detectable. The hierarchy is not a best practice — it is the physics of how errors in complex systems manifest.

Skipping a ring defers its class of errors to the next ring, where they are more expensive to find and fix — or to production, where users pay the cost.

The Agentic Era

AI coding assistants change the problem in one direction: they dramatically increase the volume of code produced per unit time. The rate of individual-level output —

functions written, modules built, tests generated — goes up substantially. What doesn't change: the physics of how errors manifest at each ring.

An orchestra that hired three times as many musicians but rehearsed the same amount would not perform better. More musicians means more pairwise interactions, more potential for ensemble errors, more that can go wrong when sections play together for the first time. The rehearsal requirement scales with the number of interacting parts — not with the quality of the individual parts.

The same is true in software. More code means more surface area for integration errors. The interaction space grows combinatorially. An AI agent that generates a new module adds not just one new component but N new relationships between that component and everything else in the system. Each relationship is a potential integration failure. None of them will be visible until the integration ring.

THE AGENT'S POSITION IN THE ENSEMBLE

Three frames help clarify what an AI agent is, in the orchestra context. They're not mutually exclusive — they describe different dimensions of the same limitation.

The Deaf Virtuoso. The agent excels at its individual part but cannot adjust to the ensemble because the ensemble is not present during generation. A trumpet player with perfect pitch who never adjusts volume to blend — not because of incompetence, but because the orchestra isn't there when the practicing happens. The adjustment requires the ensemble to be present. The integration test is when the ensemble is present.

The Section Playing Last Season's Score. The agent's context is a snapshot. Other parts of the codebase have changed since that snapshot was assembled. The agent plays what its score says — a score that was accurate when it was given, but may not match what the rest of the system is currently doing. The integration test is when all sections play together from their respective scores, and divergences become audible.

The Rehearsal Phantom. The agent generates code assuming ideal conditions: the API it calls is always available, the latency is within bounds, the data is well-formed. It cannot observe runtime failures during generation — just as a musician cannot hear

the audience's coughs before the performance. The code works in the conditions imagined. Only running it against the actual system reveals the gap.

None of these limitations are defects. They are design characteristics of how AI generation works. Their implication is structural: every ring that involves the agent's code interacting with the actual system can surface errors the agent cannot anticipate during generation. The quality of the agent's individual output does not reduce the necessity of integration validation.

THE SPECIFIC FAILURE MODES

Integration debt. Code accumulates at generation velocity; integration validation doesn't automatically scale with it. The gap between "written" and "validated at integration" is the debt. The dangerous property: its cost scales combinatorially. If N changes have been made without integration validation, the interaction surface is proportional to N^2, not N. Each additional unvalidated change makes the eventual integration more expensive.

Score drift. Agentic generation is fast enough that architectural inconsistency accumulates before anyone catches it. The agent generates a module handling a concern that belongs to a different service — because the prompt scope was ambiguous. It uses a pattern the team deprecated last quarter — because its context snapshot doesn't include that decision. Each individual piece is well-written. The piece doesn't fit the score. Section rehearsal and full-orchestra rehearsal exist precisely to catch this — but only if they happen.

The test coverage illusion. Agents write tests. The tests confirm that the code does what the agent thought it should do. They don't confirm that what the agent thought it should do is what the system needs it to do. The agent's tests are tests-after by construction — the agent knows what the code does and tests against that. Integration tests are the only ring that catches whether what the code does matches what the system needs.

Practices and Failure Modes

Each rehearsal ring is built from specific practices. Naming them matters — "write more tests" is not a practice. What follows is the named vocabulary.

AT THE INDIVIDUAL PRACTICE RING

Test-Driven Development. Writing the test before writing the code forces articulation of what the unit should do before implementing it. This distinction is not stylistic. Developers reading their own code read what they meant to write, not what they wrote. A test written after the implementation is filtered through the writer's existing understanding. A test written before is written against the intended behavior — independently of the implementation. These produce different tests, and the tests-first version catches more discrepancies between intent and implementation.

For agent-generated code: when an agent writes code, any tests it also writes are tests-after by construction. The TDD discipline applied to agentic coding means: write the tests (the specification) before invoking the agent. The agent generates code against a specification it can verify against. This constrains the agent to explicit requirements rather than invented ones.

Static analysis and type checking. Catches errors visible in isolation — type mismatches, unreachable code, known-dangerous patterns — before any test runs. The individual musician's wrong notes, audible in solo practice without the ensemble present. In agentic workflows, type inconsistencies in generated code are caught immediately by a type checker before they become integration errors.

AT THE SECTION REHEARSAL RING

Integration tests with real dependencies. Testing with actual databases, message queues, or external APIs instead of mocks. Mocking is a useful tool for speed and isolation. It is also a trap.

Mock drift is what happens when a mock diverges from the real behavior it was written to simulate. The mock was accurate when written; the real service changed; the mock wasn't updated. The test passes. The integration fails. A rate-limiting

behavior that the mock doesn't simulate. An error format that changed in the real service. A schema evolution that the mock doesn't reflect. The mock that passes when the real service would fail is not just slower to catch the error — it actively prevents the catch. Use mocks for unit tests; never for integration tests.

Contract testing. Explicit, machine-readable verification that a service's API matches what the consuming services expect — like a musical score outlining when instruments enter and exit, preventing ambiguity before rehearsal begins. When the implementation diverges from the contract, the build fails.

NAMED FAILURE MODES

Teams need vocabulary for what they're doing wrong. These names are from the research.

ECHO CHAMBER

Skipped integration tests. False confidence from tests that don't reflect actual service interactions. The individual parts sound right in isolation; the ensemble hasn't played together.

WALTZ WITHOUT SHOES

Skipped dress rehearsal. Deployment steps tested for the first time in production, amplifying the cost of failure. Something is technically executing under conditions that will cause immediate pain.

TUNING ONLY AT THE CONCERT

Skipped system tests. Production becomes the first real test environment. The audience hears the instruments being tuned during the performance.

WHAT GETS SKIPPED AND WHY

The rationalizations are specific. Naming them is half the defense.

"We'll add tests later." The cost: code accumulates, the mental model degrades, the tests written later are weaker than tests written at time of implementation. "Later"

rarely fully arrives.

"The integration tests are too slow." The cost: errors that take minutes to catch in a CI run take hours or days to debug in staging. The solution to slow integration tests is to make them faster — parallelization, containerized dependencies — not to stop running them.

"Staging is basically production." "Basically" is the problem. The errors that staging catches exist precisely because staging is not quite production: configuration differences, data volume, traffic patterns. The value of staging is in those differences.

"The AI generated this, it's probably fine." The agentic rationalization. Individual quality doesn't eliminate ensemble necessity. The quality of the part doesn't make section rehearsal optional.

"We'll catch it in monitoring." Production as the first real test. The cost: users experience the error before it's caught; incident response consumes engineering capacity; the fix requires a hotfix deployment with all the rings that were skipped.

CI/CD AS CADENCE ENFORCEMENT

Continuous integration's primary value is not automation — it is enforcement of cadence. Without CI, teams choose when to rehearse. With CI, the rehearsal happens as a structural consequence of the development workflow.

The mapping: every push triggers unit tests (individual practice ring, automated); every pull request triggers integration tests (section rehearsal, required before merge); every merge to main triggers system tests and staging deployment (full orchestra, automatic); before production, pre-production validation (dress rehearsal, required). The discipline that individual developers might exercise inconsistently is made structural. The section rehearsal happens on every PR, automatically.

The Score

Every musician in an orchestra has the same score. The shared document is the reference against which every part is measured, every interpretation checked, every divergence identified.

Software has the equivalent. In the agentic era, making it explicit has become non-optional.

Architecture Decision Records (ADRs) document the decisions that shaped the system — not just what was chosen but why, and what would change if requirements changed. Written at decision time, not retrospectively. An ADR is not documentation of what was built; it is documentation of why, which is what allows any reader — or agent — to recognize when the current implementation has diverged from the intent.

API contracts are explicit, machine-readable specifications of how services interact — OpenAPI, protobuf, JSON Schema. These are the score for inter-service interaction: what both the providing service and the consuming service are verified against. Automated contract testing fails the build when the implementation diverges from the contract.

Data models are canonical definitions of the data structures the system operates on. When a service generates a user record, what are the fields? What are the types? What are the constraints? If implicit, services develop independent interpretations that diverge.

The maintenance discipline: update the contract before updating the implementation. A score that is not maintained is a liability. An API contract that doesn't match the current implementation fails tests — which is good — but if contracts aren't updated when the implementation changes, the tests stop catching drift.

In the agentic era, the score serves an additional function: it constrains the agent's output toward architectural consistency. An agent given the current API contract generates code that conforms to it. An agent working from implicit architectural knowledge generates code that may or may not conform — and at generation velocity, the divergence accumulates before anyone catches it.

Score fragmentation is what happens when agents work from an inconsistent or absent shared reference. Without a constrained shared vision, agents optimize for inconsistent goals. Not through incompetence — through partial context. The score is the mechanism that keeps multiple agents playing the same piece.

What It Feels Like

The argument above is structural. This section is phenomenological: what it actually feels like from the inside to make these decisions, and to live with the consequences.

SKIPPING THE DRESS REHEARSAL

The decision doesn't feel like a decision. It feels like a practical response to a real constraint. The decision didn't make itself in a meeting. It unraveled over Slack. "We'll merge and monitor" became the rallying cry. The deadline is Friday. Staging has been "mostly ready" all week. The CI suite passed. The 10% of confidence you'd gain from a staging run doesn't feel like a risk when the alternative is telling the stakeholder the release slips another week.

So you don't decide to skip the dress rehearsal. You decide to ship.

What follows has its own texture. The Slack ping at 11:23pm. The dashboard showing error rates climbing. The cold laptop reopened in the dark. The technical work of debugging it is fine — developers are good at debugging. What isn't fine is the meeting that comes before the debugging. The one where someone asks "did this go through staging?" and the room understands that the answer is going to be no.

There's a silence in that moment that means something specific. It isn't anger. It's recognition. We've been here before. We will be here again. We know why and we keep doing it anyway.

The specific texture of the incident itself: it's not the frustration of a hard problem. It's the recognition of a specific kind of avoidability. You're debugging something you've

already made impossible to catch. The integration test that would have shown you this at 10am on Tuesday doesn't exist. The staging environment that would have shown you this three hours before deploy was skipped. You are now discovering, in production, with users watching, what the rehearsal would have found.

THE FIRST TIME IT PASSES

The success scene is quieter, which is why it doesn't get the post-mortems and the Slack threads. But it has a specific quality.

You've been running the integration rings throughout a major refactor — not just at the end, but as you went, each time you changed a contract, each time you modified a service boundary. The suite has found three real bugs during the refactor. It's not green because it was written to be easy to pass.

When the full suite goes green, the feeling is not relief. Relief implies you were worried. You weren't worried — you had been running the rings throughout. The feeling is closer to confirmation. The confidence didn't come from the tests themselves. It came from the rhythm. The ritual of rehearsing every merge. The knowledge that the ensemble had practiced together. The work you did was the work. The confidence you're carrying into staging is earned, not hoped for.

THE AGENTIC SEAM

There is a phase in working with AI coding assistants where the output feels like pure possibility. You produce in two hours what used to take a day. The velocity is real — that's what makes it seductive. The individual output is good code: it compiles, it passes unit tests, it does the thing you asked.

Then the integration.

Everyone's part is correct. The piece doesn't fit. Module A assumes a list; Module B returns a single item on success and null on failure. Module A assumes the operation succeeds silently; Module B raises on error. Both are internally consistent. Neither is wrong. They're playing slightly different versions of the score. And the work now is not

to fix bugs — it is to figure out, retroactively, what the piece is supposed to be, and then to fix all the modules that assumed a different piece.

The rush inverted — the fast build becomes the slow fix. The velocity of the generation phase is real. The cost of the integration phase is also real. They are not traded against each other. They are sequential.

The teams that figure this out flip the order. Don't generate until you have the integration test the generated code must pass. The test is the specification. The agent generates against the specification. The confidence doesn't come after — it is built into the process.

PAIR PROGRAMMING AS CHAMBER MUSIC

There is a specific thing that happens in pair programming that doesn't happen in code review, and it is a matter of timing.

In code review, the reviewer reads the code after it exists. In pair programming, the navigator speaks before the code exists. "Wait — what does this return when the user doesn't exist?" The driver didn't know they were making an assumption. The navigator's question doesn't change the code. It changes what code gets written.

The bug that never exists. The error erased before it ever exists as text. The catch happening in the moment before the note is struck, spotting a discord in the intention before it takes shape in the code. This is what the chamber musician does for the other player: not correcting what was played, but catching the unplayed note before it's written into the score.

The Conductor

As agents handle more of the individual implementation work, the developer's primary activity shifts. Less time writing notes; more time ensuring the notes fit the piece.

The conductor's job is not to play an instrument. It is to hold the full score — to know the piece well enough to hear when something is wrong. The conductor who has studied the Beethoven 7 for years hears the first violin play a note and knows it doesn't fit — before analyzing it, before identifying what it is, in the same way a native speaker hears a grammatical error before being able to explain the rule.

The developer who knows the architectural intent reads the agent's output and knows it doesn't fit. The module placed in the wrong service. The deprecated pattern, used again because the agent's context didn't include the deprecation. The data model choice that's locally clean and globally inconsistent. These are audible to the developer who holds the piece. To the developer who doesn't hold the piece — who hasn't read the ADRs, who is working from implicit architecture — the same output reads as plausible. Clean code, tests passing, nothing obviously wrong.

The error won't surface until integration. Until the section plays together.

The conductor role requires knowing the piece, not just the part. In the agentic era, knowing the architecture — not writing the implementation — is the critical skill. Holding the score is what makes the rehearsal structure work.

> *The performance is the rehearsals, accumulated. You don't know how the deployment will go from the deployment itself — you know from the rehearsals. When the rings were run, and the tests found things, and the findings were fixed, the deployment is the delivery of confidence already built. When the rings were skipped, the deployment is where you find out what the rings would have found.*

The professional orchestra does not treat dress rehearsal as optional overhead. The professional software team does not treat its equivalent as optional overhead. The discipline is the unseen thing that makes the visible thing — the reliable delivery, the confident deployment, the feature that ships and works — possible.

In the agentic era, this is more true, not less. More output means more to integrate. More integration surface means more need for the rings. The velocity is real. The physics hasn't changed.

Artilect US LLC — Texas

hello@artilect.us