sogeti labs
Part of Capgemini

The role of Architecture in Agentic Software Engineering

Authors - Mike Buob & Kevin Benner, PhD, Sogeti part of the Capgemini Group

Executive summary

Autonomous and semi-autonomous software agents change the operating model of software delivery and thus shifts the role that architecture plays. The purpose of architecture is no longer "How do we help humans understand and coordinate?" It becomes "How do we keep many concurrent actors—humans and agents—moving fast without diverging, coupling, or re-deciding fundamentals?" In an agentic environment, ambiguity is not merely inefficient; it is unsafe. In agent driven delivery, unclear boundaries produce architectural drift, integration churn, and compounding rework.

This white paper presents a practitioner-oriented approach to defining architecture that supports Agentic Software Engineering in the context of enterprise-class development, where architecture goes beyond explaining to humans how a system is organized and instead constrains behaviors, stabilizes decisions, and enables safe parallelism by agents without requiring constant human coordination.

We introduce six complementary architectural views that, together, support organizational scale and execution speed:

1. **Capability/domain architecture** defines the unit of parallelism (vertically sliced, independently owned capabilities).
2. **Interface & contract architecture** replaces coordination with machine-readable compliance (APIs/events/schemas/versioning rules).
3. **Decision architecture** prevents re-litigation (locked vs. open decisions, ADR governance).
4. **Execution architecture** enforces intent mechanically (repo topology, CI/CD gates, branching policies, permissions).
5. **Technical architecture** standardizes the "how" via guardrails, not shared implementation layers.
6. **Experience architecture** preserves coherent journeys and UX intent while leaving execution decentralized.

Using a CRM system as a running example, we show how capabilities like Customer, Sales, and Activity can be developed independently by multiple agents while staying coherent at the system, data, and experience levels. The central test is practical: Can a newly introduced agent pick a capability, read the constraints, and ship production-grade output without asking clarifying questions? If not, the architecture is incomplete.

1. The Purpose of Architecture in Agentic Software Engineering

In traditional delivery, architecture largely exists to help humans coordinate: it explains how the system works, clarifies responsibilities, and provides shared patterns so teams can align through conversation. In agentic delivery, the architecture needs to be more precise, providing a set of constraints across the project. When many human and AI agents execute work in parallel, the limiting factor is no longer "shared understanding"—it is uncontrolled interpretation. Agents do not slow down when something is unclear; they fill

in gaps. At machine speed, those gaps become divergent implementations, unintended coupling, and "helpful" refactors that quietly re-litigate foundational decisions.

This is why the purpose of architecture changes. In an agentic environment, architecture is not primarily a descriptive artifact; it is an execution control system. Its job is to convert intent into constraints that are stable, machine-consumable, and enforceable. Put differently: architecture must eliminate ambiguity up front, because ambiguity is no longer a coordination cost—it is an operational risk. If an agent can interpret a boundary in two plausible ways, you should assume two agents will implement both, and your system will fracture along the seam.

Agentic Software Engineering makes this shift explicit by separating authority from execution:

- **Humans** define outcomes, domains, journeys, and decisions that must not drift.
- **Agents** execute within those boundaries at scale and with high autonomy.

For that separation to work, architecture becomes the bridge between *Model* and *Delivery*: it is where intent is "compiled" into rules that delivery can safely run. This is also where traditional architectural failure modes are addressed head-on. Layered ownership, shared services, and informal coordination—reasonable strategies for human teams—become serializing bottlenecks and coupling amplifiers under agentic parallelism. Any shared layer turns into a queue; any "we'll align later" turns into inconsistent reality; any unstated decision becomes a default decision made repeatedly and differently.

So, the purpose of architecture in Agentic Software Engineering is simple but demanding:

Architecture exists to enable safe parallel execution without continuous coordination.
It stabilizes decisions, defines boundaries, and enforces compliance so that humans retain intent and agents deliver at scale—without chaos.

A practical way to tell whether you've achieved that purpose is to apply the agent-readiness lens early: a newly introduced agent (or developer) must be able to pick up a capability, read the architectural constraints, and produce production-ready output without asking clarifying questions. If questions are asked, then the architecture is still acting like documentation—informative, but not executable. In agentic delivery, architecture must behave less like a diagram and more like an operating system: defining what is allowed, what is constrained, what is locked, and how compliance is verified automatically.

2. The six-view architecture model

View 1: Capability/Domain Architecture — define the unit of parallelism

Purpose: establish vertical slices that can be built independently.

Capabilities are **bounded contexts** turned into delivery units. Each capability owns:

- application logic
- its operational data (often a dedicated schema or database)
- capability-specific UI (screens/components owned by that capability)

- tests and operational telemetry

CRM example capabilities:

- Customer (customer profile, status, identifiers)
- Contact (contacts, roles, communication preferences)
- Sales (opportunities, pipeline stages, quotes)
- Activity (calls, notes, tasks)
- Reporting (analytics projections, KPI aggregation)

The rule is simple but strict: **Only capabilities are optimized for parallel development.** Everything else exists to constrain them.

Practitioner guidance:

- Prefer *fewer, stronger* capabilities over many thin services.
- Ensure each capability is "shippable" without cross-capability code changes.
- Treat capability boundaries as the primary coordination elimination mechanism.

View 2: Interface & Contract Architecture — replace coordination with compliance

Purpose: enable independent teams/agents to integrate without conversation.

In agentic delivery, "integration" must become a contract discipline. Contracts include:

- API specs (OpenAPI/REST, gRPC, GraphQL—whatever you standardize)
- event schemas (AsyncAPI, JSON schema, protobuf)
- data product schemas (if you expose analytical views)
- versioning and compatibility rules

CRM example event contract (illustrative)

{ "eventType": "CustomerCreated",

"customerId": "string",

"occurredAt": "timestamp",

"source": "customer",

"schemaVersion": "1.0"}

Practitioner rules that matter in agentic environments:

- Events are immutable; corrections happen via new events.
- Schema evolution is additive by default; breaking changes are versioned.
- "Consumer-driven" compatibility tests are automated in CI – Continuous Integration.

The practical goal is that an agent can implement or consume an interface by reading the contract, not by asking the owning team what they "meant."

View 3: Decision Architecture — prevent re-litigation and drift

Purpose: freeze what must not be re-decided; define autonomy where allowed.

Agents are excellent at exploration, refactoring, and optimization. They are also relentless about "improving" things you intended to keep stable. Decision Architecture solves this by making key decisions explicit and classifying them:

- **Locked decisions** (non-negotiable; changing requires human approval)
- **Open decisions** (agents can choose within defined boundaries)

Practitioner techniques:

- Use lightweight ADRs, treat them as policy inputs—not narrative records.
- Put locked decisions where agents cannot miss them (repo root + build-time enforcement).
- Define "decision zones" per capability: what's open locally vs. locked globally.

View 4: Execution Architecture — enforce intent mechanically

Purpose: ensure architecture is complied with by tooling, not trust.

Execution Architecture turns architecture into **operational reality**:

- repository topology
- code ownership boundaries
- CI/CD gates and policy checks
- branching rules and release workflows
- agent permissions and safe sandboxes

Core principle: **reduce shared failure modes.**
If a single pipeline, shared library, or shared deployment gate can block everyone, you have reintroduced serialization. Guardrails should be reusable; runtime components should remain capability-owned where possible.

Execution Architecture is also where you define "agent lanes":

- what an agent can change without review
- what requires human approval
- what requires additional verification (security, compliance, data)

View 5: Technical Architecture — standardize through guardrails, not shared layers

Purpose: achieve consistency without centralizing implementation.

Organizations often respond to scale by building shared layers ("common services," "enterprise UI," "unified data access"). In agentic delivery, shared layers frequently become coordination magnets.

Instead, Technical Architecture should provide:

- approved technology stacks and versions
- reference implementations (templates, starters)
- security patterns (authN/Z, secrets, key management)
- observability standards (logging, tracing, SLOs)
- policy-as-code checks (dependency rules, linting, SAST, IaC controls)

The important nuance: these are **guardrails** and **starter kits**, not centralized runtime dependencies that every capability must change together.

View 6: Experience Architecture — centralize intent, decentralize execution

Purpose: keep the product coherent while delivery is parallel.

Experience Architecture is where humans retain strong control because experience is where value is perceived. But it should not become a bottleneck by owning every screen.

It should define:

- personas and accessibility standards
- journeys and cross-capability flows
- design system tokens/components (consumed, not rewritten)
- UX policies (navigation patterns, error states, content tone)

CRM example journey: "Close a deal" – Journey steps to "close a deal" and what capabilities in which they are realized.

- View customer → Customer capability
- View contacts → Contact capability
- Update stage → Sales capability
- Log call → Activity capability

Journeys orchestrate the intent; capabilities implement their pieces independently. This keeps the system feeling like "one product," even though it is delivered as multiple parallel slices.

3. How the six views work together (and why you need all of them)

Think of the six views as a layered control system:

- **Capability Architecture** defines *where* development parallelism is allowed.
- **Experience Architecture** defines *what coherent value* looks like.
- **Contract Architecture** defines *how* capabilities connect without coordination.
- **Decision Architecture** defines *what must not drift*.
- **Technical Architecture** defines *approved ways of building*.
- **Execution Architecture** defines *how rules are enforced continuously*.

Only Capability Architecture is optimized for execution parallelism. The other five are designed to make that parallelism safe, coherent, and repeatable.

If you omit a view:

- No contracts → agents "agree by assumption," and integration fails.
- No decision locks → agents refactor foundations endlessly.
- No execution enforcement → architecture becomes optional.
- No technical guardrails → inconsistency explodes, operability suffers.
- No experience intent → UX fragments into a patchwork.

This is why agentic architecture is not "more documentation." It is a different kind of system: one that replaces coordination with constraints.

4. Practitioners' blueprint: how to define the architecture (a practical sequence)

A useful sequence for practitioners is:

1. **Start with journeys and outcomes (Business/Model)**
   Identify 3–5 core journeys that represent value.
2. **Derive capabilities from journeys (Capability/Domain view)**
   Use bounded-context thinking: ownership, language, data, invariants.
3. **Define contracts at the seams (Interface/Contract view)**
   Prefer events for state changes and APIs for queries/commands where needed.
4. **Lock decisions that would cause widespread churn (Decision view)**
   Capture the minimum set of "global invariants."
5. **Design repo and pipeline topology to preserve independence (Execution view)**
   Make capability isolation the default; automate compliance checks.
6. **Publish guardrails and templates (Technical view)**
   Make "the right way" the easiest way.

7. **Codify experience intent (Experience view).**
   Centralize tokens, accessibility, navigation policies; decentralize screens.

Done well, this sequence produces an architecture that agents can execute against deterministically.

5. Human and agent responsibilities

In agentic delivery, authority remains human-owned; execution becomes decentralized.

Humans

- define capabilities and boundaries
- own journeys and experience intent
- lock decisions and approve exceptions
- set policies and risk controls (security, compliance)
- approve ADRs when decisions cross boundaries

Agents

- implement capability code within constraints
- generate tests, documentation, and telemetry
- refactor locally within "open" decision zones
- enforce contracts by producing compatibility tests
- propose ADRs when change is genuinely needed

The aim is not to eliminate governance; it is to make governance **machine-enforceable,** so it doesn't become a throughput limiter.

6. The agent-readiness test (the only test that matters)

A system is architected for agentic delivery when a newly onboarded agent (and developer) can:

1. select a capability
2. locate the boundaries, contracts, and locked decisions
3. implement a meaningful change
4. pass all enforcement gates
5. ship to production-grade quality
   without asking clarifying questions

If agents still have to ask, "Where does this code go?", "Which service owns this data?", "What's the standard for events?", or "Can I change this shared library?" – your architecture is missing constraints, not detail.

Conclusion: architecture becomes the operating system for delivery

When delivery shifts from human-coordinated teams to continuous, parallel execution by humans and agents, architecture can no longer function primarily as diagrams and guidance. It must become an execution control system: explicit boundaries, machine-readable contracts, locked decisions, and mechanical enforcement.

The six-view model reframes architecture as compliance rather than consensus:

1. **Capabilities** define safe parallel work.
2. **Contracts** replace coordination.
3. **Decisions** prevent drift.
4. **Execution mechanics** enforce intent.
5. **Technical guardrails** preserve consistency without centralizing delivery.
6. **Experience intent** keeps the product coherent.

With this model, organizations convert scale into speed: autonomy becomes safe, parallelism becomes sustainable, and agents become a reliable force multiplier rather than a chaos accelerator.

In agentic software engineering, architecture is no longer a backdrop to delivery.
It is the mechanism that makes delivery possible on a scale.

sogeti labs
Part of Capgemini

About Sogeti

Sogeti is a leading provider of technology and engineering services. Sogeti delivers solutions that enable digital transformation and offers cutting-edge expertise in Cloud, Cybersecurity, Digital Manufacturing, Digital Assurance & Testing, and emerging technologies. Sogeti combines agility and speed of implementation with strong technology supplier partnerships, world class methodologies and its global delivery model, Rightshore®. Sogeti brings together more than 25,000 professionals in 15 countries, based in over 100 locations in Europe, USA and India. Sogeti is a wholly-owned subsidiary of Capgemini SE, listed on the Paris Stock Exchange.

Learn more about us at

www.sogeti.com