Just-in-Time Component Specifications for an Agentic Software Platform

1. The Architecture of Just-in-Time Specification

The engineering of an agentic software platform imposes uniquely rigid constraints on how architectural intent is communicated. Unlike conventional software development, where human engineers rely on intuition to bridge contextual gaps, agentic systems require absolute determinism at the specification layer to function reliably.[1] Designing a just-in-time component specification system requires navigating a precise equilibrium to avoid two catastrophic, diametrically opposed failure modes: under-specification and premature specification.

Failure A, characterized as under-specification, emerges when implementation commences directly from a high-level architectural roadmap lacking explicit detail concerning contracts, data authority, state transitions, trust boundaries, failure handling, and operational requirements. In an agentic environment, these missing constraints prompt coding agents to hallucinate system architecture during implementation, silently inventing state transition models, and assuming unintended data authority.[1] This generates code that appears superficially functional but violates foundational architectural invariants, leading to systemic instability and unmanageable integration debt.

Failure B, characterized as premature specification, occurs when granular implementation choices are frozen artificially early. As established in lean software development methodologies, committing to technical details before upstream contracts stabilize or earlier platform stages produce operational evidence results in excessive rework and false certainty.[3] Decisions made too early in a project carry substantial risk, often leading to architectural models that must be discarded once reality diverges from preliminary assumptions, thereby creating premature technology lock-in.[5]

To reconcile these opposing risks, this research establishes an operational model built upon the lean principle of the "last responsible moment" (LRM). The LRM is a deliberate risk avoidance technique wherein decisions are deferred until the cost of not deciding exceeds the cost of making the commitment.[5] This methodology ensures that the agentic platform is specified deeply enough to enable reliable implementation, testing, review, and operation, but not so early that unverified assumptions harden into restrictive architecture.[7]

The resulting framework operationalizes a continuous pipeline: Platform vision and architectural invariants dictate requirements and Architectural Decision Records (ADRs). These artifacts subsequently define upstream contracts and threat boundaries, culminating in a rigorous specification readiness assessment.[8] Only upon passing this assessment does the formal component specification materialize, directly generating bounded implementation tasks. Implementation discoveries then feed back into the specification, ensuring the versioned document remains an immutable source of truth.[1]

1.1 Foundational Hypotheses and Principles

The architecture of an agentic platform requires the rigorous validation of foundational hypotheses regarding technical documentation and system authority.

Principle	Supporting Evidence	Limitations	Expected Operator Burden	Invalidation Condition
Master Architecture Limits	Master specs must define invariants but avoid full future component definition.[4]	Does not apply to monolithic legacy migrations where all state is pre-existing.	Low. Requires maintaining a centralized invariant list.	Widespread violation of invariants by independent components.
Incremental Specification	Specs must be produced incrementally at the LRM to avoid waste.[3]	Requires robust dependency tracking.	Medium. Requires continuous readiness monitoring.	High rework rates in specified but unimplemented components.
Upstream Contract Prerequisite	Stable schemas are mandatory for detailed downstream specs.[3]	Highly experimental, self-contained prototypes are exempt.	Low. Enforced via CI schema checks.	Downstream agents hallucinating interface payloads.
Authority Prerequisites	Security and data-authority must be known before implementation.[9]	None. This is a non-negotiable hard boundary.	High. Requires manual security review.	Silently introduced vulnerabilities or split-brain data states.
Bounded Unresolved Decisions	Local reversible implementation choices may remain open,	Difficult to strictly define "local" vs "public" programmatical	Medium. Requires subjective architectural review.	Local decisions causing breaking changes to external

	while public contract decisions block approval.[4]	ly.		consumers.
Fixture as Proof	Acceptance tests and realistic fixtures prove the boundary is understood.[1]	Generative AI models may require dynamic rather than static fixtures.	High. Requires engineering realistic mock data.	Implementation completes but fails immediate integration testing.
Normative Distinction	Specs must use RFC 2119 keywords to distinguish mandates from preferences.[1]	Human reviewers may struggle with strict protocol language.	Low. Linter-enforced.	Coding agents ignoring narrative requirements.
Traceability and Discovery	Implementation discoveries must update the spec without rewriting historical truth.[1]	Rapid prototyping may generate excessive divergence records.	Medium. Managed via Divergence schemas.	Released code bearing no resemblance to the governing specification.

2. The Specification Hierarchy and Anti-Duplication

To prevent duplication and ensure that human operators and implementation agents access a unified source of truth, the documentation ecosystem must be stratified into nine strictly bounded layers. Explicit anti-duplication rules govern this hierarchy; information defined at a higher layer must be referenced by identifier, never copied verbatim.

Layer	Function and Content Boundary	Anti-Duplication Rule
4.1 Platform Vision	Describes strategic direction, intended outcomes, operator needs, high-level capabilities, and	Must not prescribe detailed component internals.

	exclusions.	
4.2 Master Architecture	Defines architectural style, major domain planes, sources of truth, provider-independence invariants, and major trust boundaries.	Must not define individual service contracts or schemas.
4.3 Requirements Catalogue	Defines functional obligations, NFRs, security requirements, and global release gates.	Components must link via requirement_id; narrative text must not be duplicated.
4.4 ADR Catalogue	Defines consequential choices, alternatives, evidence, and migration implications.[8]	Specifications inherit adr_id constraints; rationale remains isolated in the ADR.[9]
4.5 Schema Catalogue	Defines stable machine-readable contracts (JSON Schema, Avro) across components.[11]	Complete external schemas must not be redundantly embedded inside component specifications.
4.6 Component Specification	Defines the normative implementation contract for one bounded component.	Must focus solely on local responsibilities, deferring to the master architecture for system-wide policy.
4.7 Implementation Plan	Divides the approved component specification into ordered engineering tasks.[1]	Must not invent architecture; relies entirely on the Component Specification for constraints.
4.8 Task Specification	Defines one microscopic implementation unit (objective, target files, strict bounds, and specific criteria).	Must strictly inherit NFRs and security restrictions from the parent specification.

4.9 Runbook	Defines deployment, configuration, incident response, and restoration procedures.	Operational telemetry must map directly to the Component Specification's SLO definitions.

3. Component Taxonomy and Scope Boundaries

The definition of what qualifies as a specifiable "component" determines the granularity of the specification process. The target system requires a strict taxonomy to classify deployable artifacts. Distinctions must be drawn between an architectural component, a deployable service, a simple code module, and a configuration package. Not every repository package merits a comprehensive component specification.

Component Class	Expected Depth	Required Sections	Common Risks and Checks
State Engine	Exhaustive.	All sections. Heavy focus on formal state machines, idempotency, fencing, and transaction boundaries.[12]	Split-brain execution, silent data corruption, unhandled deadlocks.[13]
Service (Domain)	High.	Contracts, ADR links, trust boundaries, NFRs, deployment impact.	Scope creep, ambiguous authority, circular dependencies.
Provider Adapter	Moderate.	Provider-independence constraints, substitution tests, capability requests.	Vendor lock-in, unhandled upstream rate limits, leaking external schemas.
Internal Library	Low.	Inputs/outputs, NFRs, algorithm performance, unit test criteria.	Becoming a hidden source of truth, unauthorized external API calls.

Gateway	High.	Trust boundaries, security controls, capacity planning, routing schemas.	Security bypass, cascading network failures, unbounded resource consumption.
Context Compiler	High.	Deterministic evaluation rules, memory backends, integration adapter points.	Non-deterministic outputs breaking downstream consumers, prompt injection vulnerabilities.

4. The Specification Readiness Gate

A fundamental operational tenet is that a component specification must not begin merely because the component appears on a strategic roadmap. It may commence only when a deterministic, reviewable Readiness Gate establishes that sufficient prerequisite evidence exists. This gate directly combats Premature Specification.

4.1 Mandatory Readiness Dimensions

The readiness assessment evaluates fifteen mandatory dimensions.

Scope Readiness: The component must have a clear purpose, explicit non-goals to prevent feature creep, a bounded responsibility, a named owner, known consumers, and identified upstream/downstream dependencies. The gate must aggressively detect overlapping responsibilities, ambiguous authority, and components defined by technology rather than by domain responsibility. A component is automatically blocked if its scope remains overly broad (e.g., "handle all agent intelligence").

Upstream Contract Readiness: An upstream contract is deemed "stable enough" for specification when its owner is known, its machine-readable schema exists, its lifecycle state is PROVISIONAL or STABLE, compatibility expectations are defined, and conformance tests exist.[3] While absolute permanence is not required, practical stability sufficient for the next implementation horizon is mandatory. Components depending on EXPLORATORY upstream contracts may only enter R1 (Exploratory) status themselves.

Requirement and ADR Readiness: The component must possess traceable functional requirements, NFRs, and security mandates. Furthermore, foundational Architecture Decision Records (ADRs) must be resolved. Blocking ADR topics include the source of truth, persistence technology class, authority boundaries, security models, identity models, and cross-project isolation models.[9] Unresolved choices regarding local directory structures or internal helper libraries are classified as IMPLEMENTATION_CHOICE and do not block readiness.[4]

Trust-Boundary and Data-Authority Readiness: Before specification approval, the

component must identify callers, untrusted inputs, network boundaries, and sandbox perimeters. It must mandate local threat models and reference system-level threat IDs. Equally critical is data authority: the component must be explicitly classified as AUTHORITATIVE, DERIVED, CACHE, INDEX, or TRANSPORT_ONLY to prevent multiple components from silently becoming disparate sources of truth.

NFR, SLO, and Fixture Readiness: Non-functional requirements (availability, durability, latency, recovery point) must be known. Provisional values are acceptable only if they include a baseline calibration experiment, an owner, and a review deadline. Readiness is empirically proven via realistic fixtures (sample tasks, protocol records, provider responses) which demonstrate that the component boundary is concrete enough to test.[1]

Acceptance-Test and Dependency Readiness: A component is unready if its success cannot be objectively verified. Minimum test sets (deterministic unit, schema conformance, security, failure injection) must be planned. Furthermore, every external dependency must be documented with an explicit fallback strategy and failure mode assumption.

Implementation-Horizon Proximity: The LRM dictates that a component should only enter formal specification when it is required within the immediate implementation horizon, is dependent on sufficiently stable upstream work, and is unlikely to be invalidated by imminent foundational decisions.[5] Full specifications for distant roadmap milestones must be aggressively rejected.

4.2 Readiness Levels, Scoring, and Hard Gates

A purely numerical readiness score is dangerous, as a high aggregate score might mask a critical, singular deficiency (e.g., unknown storage authority). Therefore, the system enforces a hybrid model where hard blocking conditions override any readiness score.

- **R0 (Not Ready):** Unclear purpose, missing requirements, unknown dependencies.
- **R1 (Exploratory):** Problem understood, alternatives under research, entirely unsuitable for implementation tasks.
- **R2 (Draft-Ready):** Scope bounded, key dependencies identified, some contracts provisional. Drafting may commence.
- **R3 (Review-Ready):** Contracts, ADRs, security, NFRs, and tests substantially defined.
- **R4 (Implementation-Ready):** All mandatory checks pass, unresolved decisions are strictly bounded, acceptance tests are defined.
- **R5 (Release-Ready):** Implementation validated, operational details confirmed, specification aligns perfectly with released code.

4.3 Specification Readiness Report Schema

Readiness is evaluated via a machine-readable JSON schema to allow deterministic CI validation before drafting begins.

JSON

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Specification Readiness Report",
  "type": "object",
  "properties": {
    "readiness_report_id": { "type": "string" },
    "component_id": { "type": "string" },
    "assessed_at": { "type": "string", "format": "date-time" },
    "blocking_findings": {
      "type": "array",
      "items": { "type": "string" }
    },
    "dimensions": {
      "type": "object",
      "properties": {
        "scope_status": { "enum": },
        "contract_status": { "enum": },
        "adr_status": { "enum": },
        "security_status": { "enum": },
        "data_authority": { "enum": },
        "fixture_status": { "enum": },
        "unresolved_decision_status": { "enum": }
      }
    },
    "recommended_state": { "enum": }
  },
  "required": ["readiness_report_id", "component_id", "blocking_findings", "dimensions",
"recommended_state"]
}
```

5. Normative Templates and Machine-Readable Schemas

To ensure specifications are comprehensible to both human operators and autonomous coding agents, the system mandates the strict separation of normative requirements from explanatory preferences, formalized via RFC 2119 terminology.[2]

5.1 Normative Language standard

The use of terms such as MUST, MUST NOT, REQUIRED, SHOULD, SHOULD NOT, and MAY establishes an unambiguous contract language that hardware and regulated software domains adopt by necessity.[2] In an agentic platform, this precise phrasing provides structured constraints

without rigid scripting, ensuring reliable execution while preserving the agent's localized reasoning ability.[1] A coding agent must be able to parse a document and cleanly separate mandatory architectural invariants from commentary. Thus, every normative statement must possess a unique traceable identifier (e.g., REQ-AUTH-042), while examples and assumptions must be visually isolated in distinct markdown blockquotes or discrete sections.

5.2 The Standard Component Specification Template

The normative template dictates the structure of the document, with sections rendered conditionally based on the component class (e.g., adapters require reduced templates).

- **22.1 Document Control:** Captures specification_id, component ID, target release, and links to superseded specifications.
- **22.2 & 22.3 Purpose and Scope:** Defines responsibility, system value, and explicitly owned behaviors, state, and contracts.
- **22.4 Non-goals:** Crucially, this section explicitly states deferred capabilities and intentionally unsupported use cases to physically prevent scope expansion by over-eager coding agents.
- **22.5 Authority Boundaries:** Details decisions the component may make, state it may mutate, and actions it MUST NEVER perform (e.g., "The component MAY propose an action but MUST NOT authorize it directly").
- **22.6 Contracts and Schemas:** Documents APIs, events, and validation bounds. For every contract, it defines transport, retry mechanisms, idempotency keys, and error semantics.
- **22.7 State Model and Lifecycle:** Defines states, guards, terminal states, and transition authority via a formal state machine definition.
- **22.8 Storage and Data Authority:** Declares authoritative records versus derived data. Defines transaction boundaries, consistency models, retention, deletion, and restoration rules.
- **22.9 Dependencies and Failure Domains:** Documents required vs optional dependencies, shared failure domains, degradation behavior, and strict dependency timeout policies.
- **22.10 Security and Trust Boundaries:** Maps to relevant threat IDs, defining assets, untrusted inputs, credentials, and residual risks.
- **22.11 NFRs and SLOs:** Defines measurable parameters for availability, latency, throughput, and capacity, linking each objective directly to telemetry.
- **22.12 Idempotency, Retries, and Recovery:** Mandates idempotency boundaries, differentiating between retryable and non-retryable failures, defining rollback and compensation logic.[1][2]
- **22.13 Observability and Configuration:** Defines logs, metrics, correlation IDs, dynamic reload schemas, and explicitly forbidden configuration patterns.
- **22.14 & 22.15 Versioning, Compatibility, and Provider Independence:** Defines compatibility policies, deprecation windows, and explicitly tests for provider substitution. Prohibits tight physical provider coupling in PROVIDER_NEUTRAL_CORE components.

- **22.17 Migration and Rollback:** Mandates data/contract migration paths, shadow modes, and irreversible step warnings.
- **22.18 & 22.19 Testing and Acceptance Criteria:** Every criterion must have an ID, statement, verification method, and linked fixture ID. Vague criteria such as "works reliably" are rejected by CI linters.
- **22.20 Open Decisions and Assumptions:** Logs bounded unresolved decisions with interim rules, owners, and strict revisit deadlines.
- **22.23 Implementation Decomposition:** Pre-fragments the specification into vertical slices, adapters, and test tasks, ensuring each generated task is small enough for independent agent implementation.

5.3 Component-Specification JSON Schema

To ensure automated tooling can validate the existence and integrity of these sections, the document is governed by a machine-readable JSON schema.

JSON

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Component Specification Constraint",
  "type": "object",
  "properties": {
    "specification_id": { "type": "string" },
    "component_id": { "type": "string" },
    "status": { "enum": },
    "authority_boundaries": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "action": { "type": "string" },
          "constraint": { "enum": }
        }
      }
    },
    "state_model": {
      "type": "object",
      "properties": {
        "idempotency_required": { "type": "boolean" },
        "outbox_pattern_enforced": { "type": "boolean" }
      }
    },
```

```
    "acceptance_criteria": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "criterion_id": { "type": "string" },
          "verification_method": { "type": "string" },
          "blocking": { "type": "boolean" }
        },
        "required": ["criterion_id", "verification_method", "blocking"]
      }
    }
  },
  "required": ["specification_id", "component_id", "status", "authority_boundaries",
"acceptance_criteria"]
}
```

6. Lifecycle, Authority, and Agent-Assisted Workflows

6.1 Lifecycle States and Invalidation

The specification lifecycle follows a strict state transition model: IDENTIFIED → READINESS_ASSESSMENT → RESEARCHING → DRAFT → REVIEW → APPROVED → IMPLEMENTING → IMPLEMENTED → VALIDATING → ACTIVE → SUPERSEDED → RETIRED.
Side states include BLOCKED (if a critical ADR is rejected), REVISIT_REQUIRED (if implementation discoveries mandate change), and EXPERIMENTAL. Any modification to an APPROVED specification immediately drops its state back to REVIEW, halting implementation tasks tied to that specific version.

6.2 Separation of Duties

A rigorous separation of duties prevents single points of architectural failure. A Requirements Steward initiates the assessment. The Component Owner drafts the architecture. However, an Architecture Reviewer, Security Reviewer, and Reliability Reviewer must independently approve their respective domains. Crucially, the Implementation Agent (whether human or AI) is strictly prohibited from unilaterally approving changes to normative component authority. A coding agent cannot decide it needs direct database access if the specification mandates an intermediary API.

6.3 Agent-Assisted Specification Workflow

Agents may safely assist the human operator by extracting linked requirements, identifying dependencies, assembling existing schemas, generating draft data-flow diagrams, proposing

failure case fixtures, and checking template completeness.[16]
However, deterministic evidence requirements govern this assistance. Before drafting, an agent must prove it inspected the master architecture, active ADRs, canonical schemas, threat models, and unresolved-decision registers. This is enforced via the **Specification Context Manifest**.

JSON

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Specification Context Manifest",
  "type": "object",
  "properties": {
    "spec_context_manifest_id": { "type": "string" },
    "component_id": { "type": "string" },
    "integrity_digest": { "type": "string", "description": "SHA-256 hash of all referenced source materials" },
    "context_sources": {
      "type": "object",
      "properties": {
        "requirements": { "type": "array", "items": { "type": "string" } },
        "adrs": { "type": "array", "items": { "type": "string" } },
        "schemas": { "type": "array", "items": { "type": "string" } },
        "threats": { "type": "array", "items": { "type": "string" } }
      }
    }
  },
  "required": ["spec_context_manifest_id", "integrity_digest", "context_sources"]
}
```

This manifest ensures reproducible specification drafting. Because the retrieval of decisions and constraints is field-weighted and deterministic, the same query against the same memory file produces byte-identical context arrays on every run.[9]

7. Conformance, Prematurity, and Under-Specification Detection

7.1 Quality Conformance Checks

Deterministic CI checks execute upon specification modification. These include schema validity, missing-section checks, normative-language linting, and checking for broken trace links. A strict

rule dictates that no physical provider names may appear in a PROVIDER_NEUTRAL_CORE specification, and all implementation tasks must link directly to verifiable acceptance criteria.

7.2 Detecting Premature Specification

Prematurity Findings are automatically generated when indicators suggest a specification is too early. These indicators include: upstream schemas do not exist; major ADRs remain unresolved; public interfaces are under active experimentation; no realistic fixture exists; technology selection precedes problem definition; or the document consists primarily of narrative assumptions rather than concrete evidence.[5] The finding identifies the affected sections and recommends a required deferral trigger.

7.3 Detecting Under-Specification

Conversely, under-specification is detected when implementation tasks require excessive architectural interpretation by the coding agent. Indicators include: the component owns undefined state; error behavior is unspecified; retry limits and idempotency keys are absent; trust boundaries are missing; acceptance criteria are vague; and operator impact is unaddressed. An implementation plan MUST NOT be approved while critical under-specification findings remain active.

8. Implementation Feedback and Divergence Policy

Implementation consistently reveals nuances invisible during the design phase. A just-in-time process implements a strict mechanism for incorporating implementation discoveries without invalidating historical records.[1]

8.1 Discovery Feedback Classifications

- **Clarification:** Implementation confirms the specification, but explanatory text needs improvement. Implementation continues.
- **Non-normative Detail:** A localized internal choice (e.g., sorting algorithm) is selected without changing external behavior. Implementation continues.
- **Specification Defect:** The approved specification is incomplete or contradictory. Implementation halts; the specification returns to REVIEW.
- **Architectural Change:** Implementation uncovers a systemic issue requiring changes to data authority, public contracts, or security boundaries. Implementation halts; a new ADR is required, and the readiness gate must be re-evaluated.[8]
- **New Evidence:** Tests invalidate earlier assumptions. Handled via formal amendment.

8.2 Divergence Records and Versioning

When implemented code deviates from the approved specification, a **Divergence Record** is generated. Acceptable local divergence may be permitted temporarily with a resolution deadline. However, material divergence (violating security, state, or contracts) triggers an immediate release block.

Specifications are tied strictly to code and release versions. Immutable released code versions

must map deterministically to specific semantic specification versions, ensuring the system can always answer exactly which specification governed Release X, and what architectural constraints changed between versions Y and Z.

9. Specification-to-Task Generation and Decomposition

Translating an APPROVED specification into executable code relies on a formal spec-to-task generation contract. This relies on natural language workflows embedded with parameterized inputs and strict constraints.[1]

9.1 The Task Generation Contract

Each generated task inherits the component_id, specification version, requirement IDs, allowed operational bounds (specific files/modules), and test obligations. Tasks are engineered to avoid crossing component authority, ensure dependency ordering, and prevent hidden migration work. A generated task record tracks completion evidence deterministically.

9.2 Implementation Decomposition Patterns

The decomposition pattern heavily depends on the component class.

- **Persistence Components:** Utilize *contract-first* or *walking skeleton* implementations, ensuring database schemas and state models are validated before business logic is layered.
- **Adapters and Gateways:** Utilize *adapter-first* patterns heavily reliant on mock fixtures to validate circuit-breaking and error translation before live upstream connection.
- **Complex Legacy Migrations:** Utilize the *strangler migration* pattern, dividing tasks into dual-read/dual-write operational phases.

Preferring vertical slices over horizontal layers ensures that tasks produce testable behavior rather than large, unvalidated infrastructure blocks that languish in non-deployable states.

10. Required Experiments and Deliverables

The research program demands empirical validation through eleven comparative experiments. **Experiment 1 (Premature Specification):** Drafting a component before schemas and ADRs stabilize. This baseline verifies the catastrophic rework and false certainty inherent in premature lock-in.[3] **Experiment 2 (Readiness-Gated):** Drafting the same component post-prerequisites, measuring the subsequent drop in ambiguity and rework. **Experiment 4 & 5 (Gate False Positives/Negatives):** Verifying the hybrid gate blocks a high-scoring component missing critical storage authority, while correctly permitting a low-risk adapter without imposing enterprise-level documentation burdens. **Experiment 6 (Task Generation):** Generating bounded tasks from an approved spec to assess requirement coverage, task size, and ambiguity.[1] **Experiment 7 & 8 (Divergence and Feedback):** Deliberately violating provider-neutrality contracts to verify automated divergence detection, and introducing a valid discovery to test the feedback classification loop. **Experiment 11 (Agent-Authored):** Allowing

an agent to draft a spec from a controlled Context Manifest.[1] This measures the agent's ability to trace requirements versus its propensity to invent unsupported architecture.

10.1 Key Deliverables and Repository Structure

The system requires specific deliverables hosted within a structured repository.

- just-in-time-component-specification-research.md (This underlying philosophy document).
- component-specification-template.md (Standard, adapter, and library variations).
- component-specification.schema.json, readiness-report.schema.json, context-manifest.schema.json, and divergence-record.schema.json.
- specification-readiness-gate.md (The scoring and blocking algorithms).
- specification-lifecycle.md (State transition models).

Suggested Repository Structure: component-specifications/ ├── README.md ├── catalogue.yaml ├── templates/ │ ├── standard-component.md │ └── internal-library.md ├── schemas/ │ ├── component-specification.schema.json │ ├── readiness-report.schema.json │ └── divergence-record.schema.json ├── readiness/ ├── active/ ├── draft/ ├── superseded/ ├── fixtures/ └── conformance/

11. Tooling, CI Integration, and Specification Debt

11.1 Methodological Comparisons

The LRM methodology draws upon but distinctively adapts existing frameworks. While *Design-by-Contract* excels at interface bounding, it lacks the architectural breadth required for full component lifecycles. *Docs-as-code* [10] provides excellent tooling but does not inherently enforce architectural gates. *Architecture Decision Records* (ADRs) [8] are crucial for capturing systemic choices but are insufficient for localized component tool-invocation constraints. The hybridized model proposed here synthesizes these methodologies, utilizing ADRs for global constraints, JSON schemas for structural determinism [9], and Docs-as-code for CI integration.

11.2 Continuous Integration Checks

Specifications must be validated via CI at multiple stages:

- **Commit Checks:** Validate schema integrity, valid IDs, link existence, normative-language linting, and provider-independence linting.
- **Pull-Request Checks:** Validate readiness evidence, ADR coverage, dependency ownership, trust-boundary mappings, and test coverage.
- **Implementation PR Checks:** Verify that changed code falls within the authorized component scope, tests satisfy criteria, and no material divergence exists.
- **Release Checks:** Ensure specification version binding, validate blocking criteria, and resolve or formally authorize any specification debt.

11.3 Specification Debt

Distinct from technical debt, Specification Debt tracks documentation failures that imperil agentic operation. Metrics include: components operating without approved specifications, the volume

of active divergences, the average age of divergences, the count of stale specifications linked to expired assumptions, and implementation tasks lacking bidirectional traceability. Excessive specification debt mandates a hard block on future feature releases until parity is restored.

11.4 Anti-Patterns to Reject

The framework explicitly rejects several pervasive anti-patterns:

- **Roadmap-driven coding:** Specifying every future component upfront.[5] Fails due to massive subsequent rework.
- **Template substitution:** Using a document template as a substitute for actual readiness prerequisites.
- **Unknown Authority:** Approving a specification without definitively classifying its storage authority. Fails due to silent split-brain data emergence.
- **Prose constraints:** Hiding unresolved decisions or normative constraints in unstructured narrative prose.[2] Fails because agents cannot reliably parse sentiment into deterministic execution rules.
- **Divergent history:** Rewriting specification history after implementation rather than utilizing formalized amendment and divergence feedback loops. Fails by destroying the audit trail of architectural drift.

12. Source Hierarchy and Quality Controls

To maintain rigorous standards, source material informing architectural decisions must follow a strict hierarchy: Formal software-engineering standards dictate baseline terminology.[2] Official architecture and SRE documentation provide operational bounds. Peer-reviewed research, maintainer-authored processes (like the ADR framework) [8], and production postmortems supply validation. Community discussion is strictly relegated to the discovery of failure modes, never as a primary source of prescriptive architecture.

Quality controls require distinct, independent responsibilities. The architecture, security, and reliability domains must be reviewed by separate entities. Deterministic scripts are utilized to eliminate human error in verifying schema validation, trace links, expired assumptions, and provider-specific leakage.

13. Phased Adoption Roadmap

Transitioning an organization to this just-in-time specification model requires a carefully orchestrated, five-phase adoption roadmap to prevent organizational paralysis.

- **Phase 0 — Governance Foundation:** Establishes terminology, the component taxonomy matrix, standard templates, and ownership models.
- **Phase 1 — The Readiness Gate:** Implements hard blockers, readiness reporting, upstream stability classification, and the unresolved-decision policy. Shifts culture from roadmap-driven to prerequisite-driven.
- **Phase 2 — Foundation Specifications:** Applies rigorous templates to core components (state engines, workflow orchestration, identity brokers). These demand maximum

architectural certainty and distributed systems rigor (leases, heartbeats, fencing, outbox patterns).[12]

- **Phase 3 — Spec-to-Task Integration:** Activates task generation contracts, links generated code directly to acceptance criteria, and implements basic divergence detection schemas.[1]
- **Phase 4 — Operational Integration:** Enforces threat model links, operationalizes SLOs via telemetry, and formalizes incident recovery mechanisms within the specification bounds.
- **Phase 5 — Portfolio Scale:** Implements cross-project dependency graphs, portfolio-wide readiness checks, inherited global requirements, and cross-project conformance testing.

14. Final Synthesis

The research unequivocally demonstrates that for an agentic software platform, a component specification is not a descriptive post-facto artifact; it is the deterministic control plane that dictates the behavior of autonomous coding agents. The foundational inquiries are synthesized below:

What constitutes a component requiring a formal specification? A component is any deployable service, state engine, gateway, adapter, or discrete evaluation subsystem possessing an independent lifecycle, a defined boundary, and an explicit data or operational authority. Simple internal libraries utilize heavily reduced templates, whereas highly concurrent state engines require the exhaustive baseline template.

What makes an upstream contract stable enough? An upstream contract allows downstream specification when its machine-readable schema is published, its lifecycle is marked PROVISIONAL or STABLE, and compatibility bounds are mathematically defined. Absolute feature-freeze permanence is unnecessary; sufficient reliability for the immediate implementation horizon is the optimal standard.

Which ADRs must exist, and which unresolved decisions block specification? ADRs concerning public APIs, persistence technology choices, data authority models, cross-project isolation, and threat boundaries must exist prior to specification, acting as strict hard-blockers.[9] Unresolved decisions regarding localized file-system structures, internal loop optimizations, and private helper functions are classified as acceptable implementation choices and do not block the gate.[4]

Which NFR values may remain provisional? Values for latency, throughput, and capacity may remain provisional during the DRAFT stage, provided they are accompanied by a specific baseline experimental methodology, a designated owner, and a firm review deadline tied to the release gate.

How should premature and under-specification be detected? Premature specification is programmatically flagged when a document relies on narrative assumptions rather than cryptographic links to existing ADRs, schemas, or realistic fixtures. Under-specification is detected automatically via CI validation when critical sections (e.g., failure domains, state transitions, idempotency rules) are omitted, or when acceptance criteria lack executable verification methods.

How should normative content be represented, and provider independence enforced? Normative content MUST be articulated using RFC 2119 terminology (MUST, SHOULD, MAY) with unique traceable IDs.[2] Provider independence is enforced by strictly prohibiting the inclusion of physical vendor names or proprietary error codes within any component designated as a PROVIDER_NEUTRAL_CORE.

How should specification context be assembled for agentic generation? Context is aggregated into a Specification Context Manifest, which cryptographically hashes the specific versions of schemas, ADRs, and threat models.[9] This guarantees that agentic drafting remains reproducible, structurally sound, and immune to unmonitored upstream drift.

How should discoveries feed back and divergence affect release? Implementation discoveries are categorized by severity. Minor details are accepted seamlessly; logical defects halt implementation for an immediate specification amendment; and fundamental architectural shifts require a new ADR and Readiness Gate re-evaluation. Material divergence from the approved specification generates formal Specification Debt and immediately blocks the CI release pipeline until parity is restored.

14.1 Final Execution Instruction

Executing a "last responsible moment" methodology requires strict, unyielding enforcement mechanisms.[5] By synthesizing RFC 2119 normative constraints [2], machine-readable JSON evaluation schemas, and deterministic readiness gates, the target orchestration platform guarantees that its implementation agents operate within a perfectly bounded architectural reality. Every resulting recommendation mandates the explicit definition of component class, readiness prerequisites, required evidence, blocking conditions, and validation methods. Where evidence is insufficient, the system must forcefully mark the component as NOT_READY, definitively rejecting the dangerous practice of filling architectural gaps with speculative implementation assumptions. The ultimate outcome is an agentic platform insulated from both the hallucinations of under-specification and the brittle, wasteful rework of premature commitment.

Works cited

1. Introducing Strands Agent SOPs – Natural Language Workflows for ..., accessed July 26, 2026, https://aws.amazon.com/blogs/opensource/introducing-strands-agent-sops-natural-language-workflows-for-ai-agents/
2. The Domain Invocation Effect: Why LLMs Generate Better Code When Shamed by Hardware Engineering Standards | by Christoforus Yoga Haryanto | Medium, accessed July 26, 2026, https://medium.com/@cyharyanto/the-domain-invocation-effect-why-llms-generate-better-code-when-shamed-by-hardware-engineering-c81d40c4da65
3. 7 Principles of Lean Software Development and How to Implement Them, accessed July 26, 2026, https://senlainc.com/blog/lean-software-development/
4. Lean Software development: The art of postponing decisions - eferro's random

stuff, accessed July 26, 2026, https://www.eferro.net/2022/08/software-development-art-of-postponing.html
5. The Last Responsible Moment - Coding Horror, accessed July 26, 2026, https://blog.codinghorror.com/the-last-responsible-moment/
6. Deferring the Last Responsible Moment - Speaker Deck, accessed July 26, 2026, https://speakerdeck.com/eoinwoods/deferring-the-last-responsible-moment
7. Lean Software Development: Decide as Late as Possible | by Edu Ferro Aldama (eferro) | Clarity AI Tech | Medium, accessed July 26, 2026, https://medium.com/clarityai-engineering/lean-software-development-decide-as-late-as-possible-a0a22dcf78d6
8. Architectural Decision Records, accessed July 26, 2026, https://adr.github.io/
9. Mneme HQ — Q&A and Glossary, accessed July 26, 2026, https://mnemehq.com/qa-glossary/
10. What is Docs as Code? Guide to Modern Technical Documentation | Kong Inc., accessed July 26, 2026, https://konghq.com/blog/learning-center/what-is-docs-as-code
11. Types and architectural styles API - tekBlueprint, accessed July 26, 2026, https://tekblueprint.org/blog/arch/essentials-api/
12. Microservices And Distributed Systems Interview Questions and Answers for .NET Developers (2026) - bool.dev, accessed July 26, 2026, https://bool.dev/blog/detail/part-9-microservices-dotnet-interview-questions
13. Heartbeats | System Design - AlgoMaster.io, accessed July 26, 2026, https://algomaster.io/learn/system-design/heartbeats
14. System Design Interview Questions by Level: Junior, Mid-Level, Senior, and Staff, accessed July 26, 2026, https://dev.to/arslan_ah/system-design-interview-questions-by-level-junior-mid-level-senior-and-staff-2m6h
15. Software Development Comprehensive Guide - Santha Lakshmi Narayana, accessed July 26, 2026, https://santhalakshminarayana.github.io/blog/software-development-comprehensive-guide
16. The Agentic SDLC: The Software Lifecycle, Rebuilt Around Agents - Port.io, accessed July 26, 2026, https://www.port.io/blog/agentic-sdlc-software-lifecycle-rebuilt-around-agents
17. GitHub - strands-agents/agent-sop: Natural language workflows that enable AI agents to perform complex, multi-step tasks with consistency and reliability., accessed July 26, 2026, https://github.com/strands-agents/agent-sop
18. Architectural decision record schema identifier format | Yoda, accessed July 26, 2026, https://utrechtuniversity.github.io/yoda/design/metadata/adr-schema-identifiers.html
19. Agentic Network Architecture and Protocol for Supporting Agent Interconnection Communication and Multi-level Inference - IETF, accessed July 26, 2026, https://www.ietf.org/archive/id/draft-chuyi-nmrg-agentic-network-inference-00.html
20. Top 96 Distributed Computing Interview Questions And Answers 2025 | TechPrep,

accessed July 26, 2026, https://www.techprep.app/full-stack/distributed-computing
21. Distributed Systems Interview Questions for Senior Engineers (2026) - Algoroq, accessed July 26, 2026, https://www.algoroq.io/interview-questions/distributed-systems/