SPOQ: Specialist Orchestrated Queuing for Multi-Agent Software Engineering

Royce Carbowitz
Independent Researcher
royce.carbowitz@gmail.com

February 18, 2026
v0.4.3

Abstract

Multi-agent AI systems show promise for automating software engineering tasks, yet existing approaches suffer from coordination overhead, quality control gaps, and limited human oversight. We introduce **SPOQ** (Specialist Orchestrated Queuing), a methodology for multi-agent software development combining three innovations: (1) *wave-based topological dispatch* that computes parallel execution waves from task dependency graphs; (2) *dual validation gates* applying quality metrics before execution (planning validation) and after (code validation) to reduce rework cycles; and (3) *Human-as-an-Agent (HaaA)* integration, where a human specialist participates in decomposition and can be consulted during execution. SPOQ uses a three-tier agent hierarchy (Opus workers, Sonnet reviewers, Haiku investigators), each selected to optimize cost-quality tradeoffs. Evaluation across six projects spanning web development, backend services, and system tooling demonstrates 1.3–5.3× speedup over sequential execution. We discuss failure modes, lessons learned, and implications for AI-native software engineering.

Keywords: Multi-agent systems, LLM orchestration, human-AI collaboration, task decomposition, quality assurance, software engineering automation

Contents

Source Code: https://gitlab.com/kenth56/spoq
Reference implementation, epic definitions, validation skills, and all case study artifacts.

1 Introduction

The emergence of large language models (LLMs) capable of generating, understanding, and reasoning about code has sparked intense interest in multi-agent systems for software engineering automation [Li et al., 2023, Hong et al., 2023]. These systems promise to decompose complex software projects into subtasks, assign them to specialized agents, and coordinate their execution toward a unified goal. Early results demonstrate that multi-agent collaboration can produce functioning software artifacts, from simple games to multi-file applications [Qian et al., 2023].

However, current multi-agent approaches face three fundamental challenges:

Coordination Overhead. Systems like ChatDev [Qian et al., 2023] and MetaGPT [Hong et al., 2023] rely on sequential role-playing or message-passing between agents, creating bottlenecks that limit parallelism. When Agent A must wait for Agent B's output before proceeding, potential speedups from parallel execution remain unrealized.

Quality Control Gaps. Most multi-agent systems lack structured validation between planning and execution phases. Agents execute plans without rigorous assessment of plan quality, leading to wasted computation when fundamental flaws are discovered late. Similarly, post-execution quality assessment is often informal or absent.

Limited Human Oversight. Fully autonomous multi-agent systems exclude human judgment from the loop, missing opportunities to leverage human expertise for task decomposition, ambiguity resolution, and quality assessment. When agents encounter edge cases or make suboptimal decisions, there is no mechanism for human correction.

1.1 SPOQ: Our Approach

We introduce **SPOQ** (Specialist Orchestrated Queuing), a methodology that addresses these challenges through three integrated innovations:

1. **Wave-Based Topological Dispatch:** We model task dependencies as a directed acyclic graph (DAG), a structure where arrows show which tasks must complete before others can begin, with no circular dependencies (think of it as a flowchart where all arrows point forward). We then compute execution *waves* (groups of independent tasks) via topological sort. Tasks within the same wave execute in parallel, while waves execute sequentially to respect dependencies. This maximizes parallelism without coordination overhead.

2. **Dual Validation Gates:** We apply structured validation (quality checkpoints with scored metrics) at two points: *before* execution (planning validation with 10 metrics) and *after* execution (code validation with 10 metrics). Each gate enforces a 95% threshold, catching quality issues when they are cheapest to fix.

3. **Human-as-an-Agent (HaaA):** A human specialist participates alongside AI agents, not as a passive observer, but as an active collaborator who decomposes tasks, validates plans, and can be consulted during execution. This bidirectional integration treats the human as a high-value agent within the system.

1.2 Contributions

This paper makes the following contributions:

- A formal framework for wave-based multi-agent orchestration that computes parallel execution waves from task dependency graphs (Section 3)
- A three-tier agent hierarchy (Opus/Sonnet/Haiku) that optimizes cost-quality tradeoffs by matching model capability to task complexity (Section 3.4)
- The Human-as-an-Agent (HaaA) paradigm for structured task decomposition through bidirectional human-AI collaboration (Section 3.5)
- A dual validation system with explicit metrics that scores both planning quality and code quality against quantified thresholds (Section 4)
- Proof-of-concept evaluation across six projects demonstrating 1.3–5.3× speedup while preserving quality above validation thresholds (Section 6)

Key Terms at a Glance

Epic: A high-level goal decomposed into atomic tasks, each with explicit dependencies, acceptance criteria, and time estimates.

Wave: A group of tasks sharing no mutual dependencies, enabling simultaneous execution by multiple agents within a single phase.

DAG: Directed Acyclic Graph, a structure where arrows represent prerequisite relationships between tasks, with no circular dependencies.

Critical Path: The longest sequential chain of dependent tasks through the DAG; determines the minimum possible project duration.

Validation Gate: A scored quality checkpoint where work must exceed defined metric thresholds before the pipeline advances to the next stage.

HaaA (Human-as-an-Agent): A bidirectional collaboration paradigm where humans participate alongside AI agents as active contributors.

PERT Estimates: Three-point time estimates capturing optimistic, realistic, and pessimistic durations to quantify scheduling uncertainty.

2 Background and Related Work

2.1 Multi-Agent Systems for Software Engineering

Multi-agent approaches to software engineering have gained momentum with the advancement of LLM capabilities. We survey three representative systems:

ChatDev [Qian et al., 2023] simulates a virtual software company with role-playing agents (CEO, CTO, Programmer, Tester) that communicate through structured chat. While effective for generating simple applications, the sequential communication pattern creates bottlenecks, since each agent must wait for prior agents to complete their turns before contributing.

MetaGPT [Hong et al., 2023] introduces standardized operating procedures (SOPs) that structure agent collaboration around software artifacts (PRDs, design documents, code). This reduces communication overhead compared to free-form chat but still relies on sequential handoffs between roles.

Multi-Agent Debate (MAD) [Liang et al., 2023] uses multiple agents that debate and refine solutions iteratively. While debate can improve solution quality through diverse perspectives, the synchronous turn-taking limits parallelism.

SPOQ differs from these systems in three ways: (1) wave-based parallel execution rather than sequential role-playing; (2) explicit validation gates with quantified metrics; and (3) structured human integration rather than full autonomy.

2.2 Task Decomposition and Dependency Management

Hierarchical task decomposition has roots in classical AI planning [Nau et al., 2003]. Modern approaches apply LLMs to generate task breakdowns:

Hierarchical Task Networks (HTNs). Classical HTN planners decompose abstract tasks into primitive actions with ordering constraints. SPOQ adapts this structure for multi-agent software engineering.

DAG Scheduling. Topological sorting of directed acyclic graphs is well-established for parallel task scheduling [Coffman Jr and Graham, 1972]. SPOQ applies these algorithms to LLM-generated task dependencies, computing wave assignments that maximize parallelism while respecting precedence constraints.

Critical Path Analysis. PERT and CPM methods identify the longest path through a dependency graph [Kelley Jr and Walker, 1959]. SPOQ uses critical path analysis to estimate minimum wall-clock execution time and identify bottleneck tasks.

2.3 Human-AI Collaboration in AI-Native Engineering

Human-in-the-loop (HITL) approaches to AI systems span a spectrum from passive oversight to active collaboration:

Prompt Engineering. Users craft prompts to guide LLM behavior but have limited ability to influence intermediate reasoning or correct course mid-execution [White et al., 2023].

AI Pair Programming. Tools like GitHub Copilot [Chen et al., 2021] suggest code completions while humans retain editing control. This inverts SPOQ's model: humans do the work while AI assists, rather than agents doing the work while humans validate.

Supervised Autonomy. Some systems allow human intervention at checkpoints [Wu et al., 2023]. SPOQ extends this with bidirectional communication: not only can humans intervene, but agents can explicitly request human assistance when facing ambiguity.

2.4 Autonomous Agent Frameworks

Early autonomous agent frameworks pioneered recursive goal decomposition and self-directed task execution, establishing foundational patterns for agentic AI:

AutoGPT and BabyAGI. AutoGPT [Richards, 2023] and BabyAGI [Nakajima, 2023] introduced recursive goal decomposition where agents autonomously break objectives into subtasks and execute them. These early systems demonstrated that LLMs could function as autonomous agents but lack explicit dependency management and formal quality gates, leading to unpredictable execution paths. SPOQ addresses these limitations through DAG-based scheduling with dual validation gates.

CrewAI. CrewAI [Moura, 2024] takes a role-based approach to multi-agent orchestration, defining agents by their personas (e.g., researcher, writer, editor) and coordinating them through a sequential pipeline. While effective for role-delegation workflows, CrewAI does not provide built-in quality validation gates or DAG-based parallel dispatch. SPOQ differs by operating at task granularity (1–4 hour atomic units) rather than role granularity, and by enforcing measurable quality thresholds at two distinct checkpoints.

LangGraph. LangGraph [LangChain, Inc., 2024] models agent workflows as directed graphs where nodes represent LLM invocations and edges define control flow. This graph-based approach enables fine-grained orchestration at the individual LLM-call level. In contrast, SPOQ's directed acyclic graph operates at task granularity—each node represents hours of focused implementation work rather than a single model invocation—and includes explicit quality validation between execution waves.

2.5 Autonomous Software Engineering Agents

A new category of autonomous software engineering agents has emerged, capable of navigating codebases, writing code, running tests, and debugging:

Devin and OpenHands. Devin (Cognition Labs, 2024) [Cognition Labs, 2024] demonstrated end-to-end autonomous task completion, while OpenHands [Wang et al., 2024] provides an open-source alternative with similar capabilities. Both excel at single-agent execution but focus on individual task autonomy rather than multi-agent orchestration. SPOQ complements such systems by providing the coordination layer for parallel execution across multiple agents.

2.6 Code Generation and Pair Programming Tools

Developer-facing code generation tools occupy a different point in the autonomy spectrum, emphasizing human-AI collaboration over full autonomy:

Aider and GPT-Engineer. Aider [Gauthier, 2024] enables conversational code editing where developers describe changes in natural language and the tool applies them to local repositories. GPT-Engineer [Osika, 2023] generates entire applications from specifications, producing directory structures and boilerplate code. Both tools operate as single agents without parallel execution capabilities or formal validation beyond user inspection.

Claude Code. Claude Code (Anthropic, 2025) marked an inflection point for AI-assisted software development, establishing the agentic coding paradigm where developers collaborate with an autonomous agent that navigates codebases, executes commands, and iterates on solutions within a persistent terminal session. Unlike completion-based tools, Claude Code operates with full project context and can orchestrate multi-step workflows autonomously, making it the catalyst that brought agentic development to mainstream adoption. SPOQ builds on this foundation: where Claude Code enables a single developer-agent partnership, SPOQ coordinates multiple such agents in parallel with structured quality gates.

GitHub Copilot. GitHub Copilot [Chen et al., 2021] suggests code completions while humans retain editing control, inverting the SPOQ model: humans perform the work while AI assists, rather than agents performing work while humans validate. SPOQ targets a different use case where agents drive execution with human oversight, enabling higher throughput on parallelizable tasks.

Positioning SPOQ. We note that Claude Code, Devin, and similar products are evolving rapidly; the capabilities described reflect their state as of late 2025. SPOQ does not compete with execution-focused agents or code generation tools. Rather, it provides an orchestration layer that coordinates multiple agents on complex projects. The key insight is that orchestration and execution are separable concerns: an agent like Claude Code excels at executing individual tasks, while SPOQ excels at decomposing projects into tasks, scheduling them for parallel execution, and validating quality at each gate.

3 The SPOQ Methodology

SPOQ orchestrates multi-agent software development through a four-stage pipeline: Epic Planning, Epic Validation, Agent Execution, and Agent Validation. We present the formal framework, key algorithms, and design principles underlying each stage.

3.1 Design Principles

SPOQ is built on five design principles that distinguish it from prior multi-agent approaches:

Design Principle 3.1 (Atomic Task Boundaries)**.** Each task constitutes 1–4 hours of focused work with one clear deliverable. Tasks are self-contained: a worker agent can complete the task without coordinating with other agents during execution.

Design Principle 3.2 (Explicit Dependencies)**.** Task dependencies form a directed acyclic graph (DAG), a diagram where arrows indicate prerequisite relationships, and no task can indirectly depend on itself. All dependencies are declared upfront, enabling static analysis of parallelism and identification of the *critical path* (the longest chain of dependent tasks that determines minimum project duration).

Design Principle 3.3 (Quality Gates Before and After)**.** Validation occurs both before execution (planning quality) and after execution (code quality). Early validation catches expensive mistakes when they are cheapest to fix.

Design Principle 3.4 (Human-AI Collaboration)**.** A human specialist participates in task decomposition, validates plans, and can be consulted by agents. The human is a high-value agent, not an external observer.

Design Principle 3.5 (Cost-Optimized Agent Selection)**.** Different roles require different capability-cost tradeoffs. High-capability agents (Opus) handle complex tasks; balanced agents (Sonnet) handle quality review; fast-cheap agents (Haiku) handle triage.

3.2 Four-Stage Pipeline

Figure 1 illustrates the SPOQ pipeline.

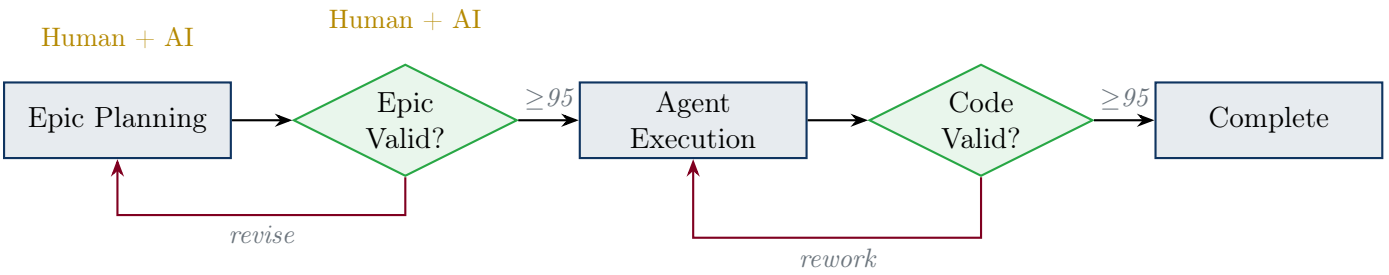

Figure 1: SPOQ four-stage pipeline with dual validation gates. Human specialist participates in planning and plan validation. Failed validations trigger revision/rework loops.

Definition 3.1 (Epic)**.** An *epic* $E = (G, T, D, S)$ consists of:

- G: Goal statement describing the desired outcome
- $T = \{t_1, t_2, \ldots, t_n\}$: Set of atomic tasks
- $D \subseteq T \times T$: Dependency relation where $(t_i, t_j) \in D$ means t_i must complete before t_j can begin

- $S = \{s_1, s_2, \ldots, s_m\}$: Success criteria for the epic

Definition 3.2 (Task)**.** A *task* $t = (id, desc, deps, files, criteria, est)$ consists of:

- id: Unique identifier within the epic
- $desc$: Implementation description with steps
- $deps \subseteq T$: Set of prerequisite tasks (work that must finish first)
- $files$: List of files to be modified
- $criteria$: Acceptance criteria for task completion
- $est = (o, r, p)$: Three-point estimate: best-case (o), expected (r), and worst-case (p) durations, borrowed from PERT project management

3.3 Wave-Based Topological Dispatch

Given a task dependency graph, SPOQ computes *waves*, groups of tasks that can execute in parallel because they have no dependencies on each other. The algorithm uses *topological sorting*, a standard graph algorithm that orders nodes (tasks) such that all dependencies appear before the nodes that require them.

Definition 3.3 (Wave Assignment)**.** A *wave assignment* $W : T \to \mathbb{N}$ maps each task to a non-negative integer such that:

$$\forall (t_i, t_j) \in D : W(t_i) < W(t_j) \tag{1}$$

Tasks in the same wave have no dependencies between them and can execute concurrently.

Algorithm 1 presents the wave computation procedure.

Algorithm 1 Wave Computation via Topological Sort

Require: Task set T, dependency relation D
Ensure: Wave assignment $W : T \to \mathbb{N}$

```
indegree[t] ← |{t' : (t', t) ∈ D}| for all t ∈ T
W[t] ← ⊥ for all t ∈ T
w ← 0
while ∃t ∈ T : W[t] = ⊥ do
    ready ← {t ∈ T : W[t] = ⊥ ∧ indegree[t] = 0}
    for t ∈ ready do
        W[t] ← w
    end for
    for t ∈ ready do
        for t' : (t, t') ∈ D do
            indegree[t'] ← indegree[t'] − 1
        end for
    end for
    w ← w + 1
end while
return W
```

Theorem 3.1 (Parallelism Bound)**.** *Let $W^* = \max_{t \in T} W(t)$ be the number of waves. The wall-clock execution time is bounded below by:*

$$T_{wall} \geq \sum_{w=0}^{W^*} \max_{t:W(t)=w} duration(t) \tag{2}$$

This bound is achieved when sufficient agents are available to execute all tasks in each wave simultaneously.

Critical Path Analysis. The *critical path* is the longest chain of dependent tasks through the graph, weighted by durations. Even with unlimited parallel resources, the project cannot complete faster than this path; it represents the irreducible sequential bottleneck. Identifying critical path tasks reveals where delays have the greatest impact. The theoretical minimum execution time is:

$$T_{\text{critical}} = \max_{\text{path } P} \sum_{t \in P} \text{duration}(t) \tag{3}$$

SPOQ reports the *speedup factor* $\sigma = T_{\text{sequential}}/T_{\text{critical}}$, comparing sequential execution time (all tasks one-by-one) to the parallelized critical path time. A speedup of 5.3x, for example, means the work completes in roughly one-fifth the time it would take serially.

3.4 Three-Tier Agent Hierarchy

SPOQ employs three agent tiers, each optimized for its role:

Table 1: SPOQ Agent Hierarchy

Tier	**Model**	**Role**	**Tradeoff**
Worker	Opus	Task execution	High capability, high cost
Reviewer	Sonnet	Quality assurance	Balanced capability/cost
Investigator	Haiku	Build failure triage	Low cost, fast response

Opus Workers. For each task t in the current wave, SPOQ spawns an Opus agent with: (1) the task specification, (2) relevant epic context, (3) completed dependency outputs, and (4) prior QA feedback if this is a rework attempt.

Sonnet Reviewers. Each completed task undergoes quality review by a Sonnet agent, which scores the work against 10 code quality metrics (Section 4). Tasks scoring below threshold are queued for rework with specific remediation guidance.

Haiku Investigators. When the build fails after a wave, a Haiku agent analyzes the error output to determine which tasks likely caused the failure. This triage is fast and inexpensive, allowing rapid identification of problematic tasks without engaging the full QA process.

Platform Independence. While our reference implementation uses Anthropic's Claude model family, the SPOQ methodology is inherently platform-agnostic. The three-tier hierarchy represents an abstract capability mapping: any sufficiently capable model family can populate the Worker (high

Table 2: Capability Tier Mapping Across LLM Providers

Tier	**Role**	**Claude**	**OpenAI**	**Gemini**
Worker	Task execution	Opus	GPT-4	Ultra/Pro
Reviewer	Quality assurance	Sonnet	GPT-4-turbo	Pro
Investigator	Build triage	Haiku	GPT-3.5	Flash

capability, high cost), Investigator (low cost, fast response), and Reviewer (balanced capability/cost) tiers. Table 2 illustrates potential mappings across providers.

The core methodological contributions, including wave-based topological dispatch, dual validation gates, explicit dependency DAGs, and Human-as-an-Agent integration, require no vendor-specific features and transfer directly to alternative platforms. Organizations can implement SPOQ using their preferred LLM provider by calibrating capability tiers to their available models and adjusting cost thresholds accordingly.

3.5 Human-as-an-Agent (HaaA) Integration

SPOQ treats the human specialist as a high-value agent integrated into the orchestration loop, not an external supervisor. Rather than simply monitoring AI agents from the sideline, the human actively participates, contributing expertise where it matters most and receiving assistance in return.

Definition 3.4 (Human-as-an-Agent)**.** The *Human-as-an-Agent* (HaaA) paradigm defines bidirectional integration, meaning communication flows both ways between human and system:

1. **Human→System:** The human participates in epic planning, validates epics before execution, and can intervene during execution.
2. **System→Human:** Agents can request human assistance when facing ambiguity, blocked progress, or decisions beyond their scope.

This bidirectional model enables *quality amplification*: the human's judgment improves task decomposition quality (reducing downstream rework), while agents' execution scales the human's productivity.

Task Decomposition. The human specialist drafts epics using hierarchical task decomposition, assisted by LLM suggestions. The human ensures:

- Tasks are appropriately scoped (1–4 hours)
- Dependencies are correctly identified
- Acceptance criteria are verifiable
- Potential risks are mitigated

Validation Participation. Before execution, the human reviews the epic alongside the automated validation. The human can approve, request revisions, or override automated assessments with justification.

Consultation Requests. During execution, agents may encounter situations requiring human judgment: ambiguous requirements, conflicting dependencies, or decisions with significant implications. Agents can pause and request human input rather than proceeding with potentially incorrect assumptions.

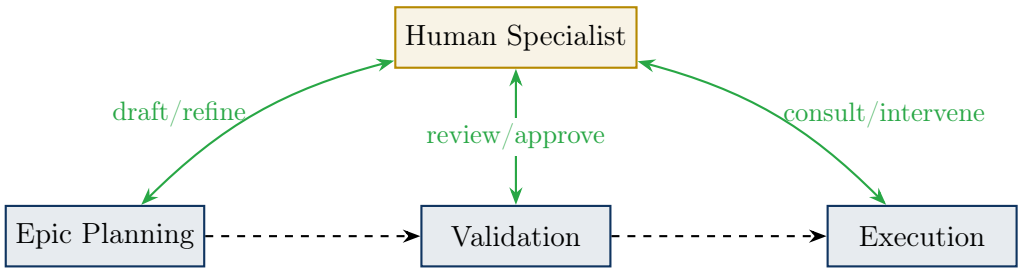

Figure 2: Human-as-an-Agent integration. Bidirectional arrows indicate two-way communication: human contributes to each stage and can be consulted by the system.

4 Validation Framework

SPOQ enforces quality through two *validation gates*, structured checkpoints where work is scored against explicit criteria before proceeding. The first gate, *epic validation*, occurs before execution and assesses plan quality. The second, *agent validation*, occurs after execution and assesses code quality. Each gate applies 10 metrics, scored 0–100, with a 95% aggregate threshold for passing.

4.1 Planning Validation: 10 Metrics

Epic validation assesses whether the plan is sufficiently clear, complete, and well-structured to enable successful execution. Table 3 summarizes the metrics.

Table 3: Epic Validation Metrics

Metric	**Question**	**Threshold**
Vision Clarity (VC)	Is the epic's goal clearly scoped?	$\geq$90
Architecture Quality (AQ)	Is the architecture diagram complete?	$\geq$90
Task Decomposition (TD)	Are tasks atomic and independent?	$\geq$90
Dependency Graph (DG)	Are dependencies explicit and acyclic?	$\geq$90
Coverage Completeness (CC)	Do tasks fully cover all success criteria?	$\geq$90
Phase Ordering (PO)	Do waves maximize parallelism?	$\geq$90
Scope Coherence (SC)	Do all tasks contribute to the epic goal?	$\geq$90
Success Criteria Quality (SQ)	Are criteria SMART?	$\geq$90
Risk Identification (RI)	Are blockers and risks mitigated?	$\geq$90
Integration Strategy (IS)	Is it clear how tasks merge and verify?	$\geq$90

Pass Criteria. An epic passes validation if:

$$\frac{1}{10}\sum_{i=1}^{10} M_i \geq 95 \quad \wedge \quad \min_i M_i \geq 90 \tag{4}$$

This dual requirement ensures both high average quality and no critically weak dimensions. A perfect score on 9 metrics cannot compensate for a failing score on the 10th.

Threshold Rationale. The 95/90 planning threshold reflects the cascading cost of planning errors: a poorly-structured task creates execution problems that propagate to dependent tasks. We observed that plans scoring below 90 on any metric required rework in over 50% of cases, while those scoring above 95 overall experienced less than 10% rework. The minimum threshold of 90 prevents "averaging out" a critically weak dimension. Appendix A.1 provides additional design rationale.

Rationale: Validate Early. Planning mistakes are cheap to fix but expensive to execute. By enforcing rigorous validation before spawning agents, SPOQ prevents wasted computation on fundamentally flawed plans.

4.2 Code Validation: 10 Metrics

Agent validation assesses whether completed work meets quality standards. Table 4 summarizes the metrics.

Table 4: Agent Validation Metrics

Metric	Question	Threshold
Syntactic Correctness (SC)	Does the code compile without errors?	≥80
Test Existence (TE)	Does new code have corresponding unit tests?	≥80
Test Pass Rate (TP)	Do all tests pass?	≥80
Requirements Fidelity (RF)	Does implementation match task specification?	≥80
SOLID Adherence (SA)	Does code follow SOLID design principles (Single responsibility, Open/-closed, etc.)?	≥80
Security (SE)	Free from OWASP Top 10 vulnerabilities (injection, auth flaws, etc.)?	≥80
Error Handling (EH)	Does code handle failures gracefully?	≥80
Scalability (SL)	Will this code scale appropriately?	≥80
Code Clarity (CC)	Is code readable and self-documenting?	≥80
Completeness (CO)	Is work fully finished (no TODOs/stubs)?	≥80

Pass Criteria. A task passes validation if:

$$\frac{1}{10}\sum_{i=1}^{10} M_i \geq 95 \quad \wedge \quad \min_i M_i \geq 80 \tag{5}$$

The per-metric floor is lower (80 vs 90) because code quality inherently involves tradeoffs that planning does not.

Code Threshold Rationale. The 95/80 code threshold is more lenient than planning because code can be iteratively improved post-delivery, and some metrics (e.g., security edge cases) are inherently harder to achieve perfectly. Rework at the code level is less expensive than re-planning an entire task decomposition; a failed task can be re-executed with targeted feedback, whereas a flawed plan may invalidate work across multiple dependent tasks.

Concise Feedback. On failure, the reviewer provides remediation guidance in ≤20 lines: specific file:line references, concrete issues, and numbered action items. This constraint respects the context budget of the orchestrator and forces actionable specificity.

4.3 Validation Cascade

SPOQ applies a *validation cascade*, a hierarchical check where the overall epic validation incorporates individual task-level assessments. Epic validation triggers task-level validation on each constituent task. If any individual task scores <95, the epic's aggregate score is capped at 85, forcing a FAIL verdict.

This prevents "carrying" weak tasks on the strength of strong planning. Every task must be individually well-specified to pass epic validation.

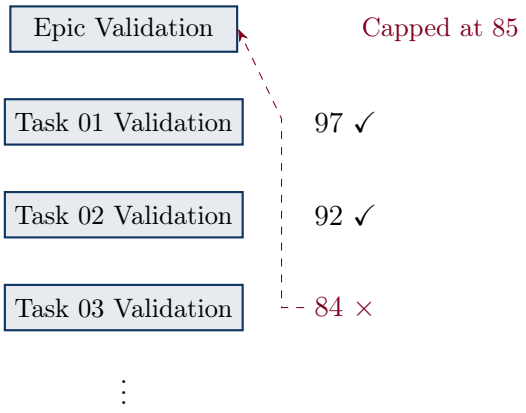

Figure 3: Validation cascade: a single task scoring <95 caps the epic's overall score, triggering failure.

5 Implementation

This section describes the practical realization of SPOQ, including the file-system representation of epics and tasks, the journal-based tracking system for agent work sessions, and the skill framework that encapsulates reusable agent capabilities. See Appendix A.3 for complete schema examples, journal entry format, and skill metadata.

5.1 Task Representation: The spoq/epics/ Directory

SPOQ represents epics and tasks as structured files in a designated directory, enabling version control, human review, and programmatic manipulation.

Epic Lifecycle. Epics follow a two-stage directory lifecycle that reflects their progression through the SPOQ pipeline. During planning, new epics are created under `spoq/epics/active/`, where they remain throughout validation and execution phases. Once every constituent task passes the agent-validation gate, the orchestrator relocates the entire epic directory to `spoq/epics/complete/`. This filesystem-level transition serves as an unambiguous completion signal: version control history preserves the move as a single commit, providing an auditable record of when each epic achieved full validation. Separating in-flight work from verified deliverables also prevents agents from inadvertently modifying artifacts that have already satisfied quality thresholds.

Roadmap Coordination. The `ROADMAP.md` file at the root of `spoq/epics/` acts as a centralized registry of all epics and their current disposition. During planning, the orchestrator appends a new entry containing the epic identifier, a brief objective summary, and an initial status of *planned*. As execution proceeds, status fields are updated to reflect transitions through *in-progress*, *validation*, and *done*. Because the roadmap records inter-epic dependencies alongside status, it enables the orchestrator to determine which downstream epics become unblocked when a predecessor completes.

Task YAML Schema. Each task file follows a standardized YAML schema with three categories of fields:

Definition 5.1 (Task Specification)**.** A *task specification* is a YAML document with the following structure:

Identity fields (`id`, `title`, `epic`) establish task context by assigning a unique identifier within the epic, a human-readable title, and a back-reference to the parent epic. **Execution control fields** govern scheduling through status tracking (`pending`/`in_progress`/`completed`), priority levels, wave assignment via the `phase` field, PERT three-point effort estimates, dependency lists referencing prerequisite task IDs, and required domain skills. **Deliverable fields** define verification criteria: `files_to_touch` enumerates all files the agent may modify, `outputs` lists tangible deliverables, and `acceptance_criteria` provides a checklist the agent must satisfy before marking the task complete. A free-form **description** field supplies structured implementation guidance using Markdown, typically containing an objective statement, step-by-step instructions with code snippets, and verification commands. See Appendix A.3 for annotated examples of each field category.

Phase Assignment. The `phase` field encodes wave assignment from topological analysis:

- Phase 0: Tasks with no dependencies (Wave 0)
- Phase n: Tasks depending only on tasks in phases $< n$

Tasks within the same phase execute concurrently. The critical path determines the minimum number of waves required.

5.2 Journal Tracking System

SPOQ employs a journal-based tracking system that records agent work sessions with structured metadata, enabling explainability, meta-orchestration, and training data generation.

Definition 5.2 (Journal Entry)**.** A *journal entry* consists of YAML frontmatter followed by Markdown sections documenting a completed work session.

The frontmatter captures machine-readable metadata: agent identity, ISO 8601 timestamps, a calibrated confidence score (0.0–1.0), session type classification, the list of modified files, and task completion counts. The Markdown body follows a standardized layout with sections for summary, work completed, changes made, issues encountered, testing results, and next steps. See Appendix A.3 for complete format examples and tooling support.

Confidence Scoring. Each journal entry includes a calibrated confidence score (0.0–1.0) reflecting the agent's self-assessment of work quality. Scores above 0.85 indicate well-tested, production-ready output; scores between 0.65–0.84 signal functional work requiring additional validation; scores below 0.65 flag experimental results with known gaps. See Table A.7 in the Appendix for the full interpretation scale.

Value Proposition. The journal system provides four categories of value:

1. **Explainability (XAI):** Every work session is documented with rationale, enabling audit trails and decision archaeology.
2. **Multi-Agent Coordination:** Timestamps and task progress enable parallel agents to avoid conflicts and build on each other's work.
3. **Knowledge Graph Construction:** Journal entries create nodes (sessions, files, tasks) and edges (MODIFIED, COMPLETED) for analysis.
4. **Training Data:** Combined with git commits, journal entries provide rich examples for fine-tuning development-focused LLMs.

5.3 Skill Framework

SPOQ encapsulates reusable agent capabilities as *skills*, self-contained modules that provide domain knowledge, workflows, and tooling for specific task categories.

Definition 5.3 (Skill)**.** A *skill* is a directory containing:

- `SKILL.md`: Instructions and metadata (required)
- `scripts/`: Executable utilities (optional)
- `references/`: Documentation loaded on demand (optional)
- `assets/`: Templates and configurations (optional)

Core SPOQ Skills. The framework includes six skills aligned with the four-stage pipeline (see Table A.8 in the Appendix for the full inventory).

Skill Invocation and Anatomy. Skills are invoked via slash commands (e.g., `/epic-planning`, `/agent-execution`) that expand into full prompts with context from the referenced epic directory. Each skill's `SKILL.md` contains YAML frontmatter (name, description) followed by structured documentation covering activation criteria, core patterns, quality standards, and integration points with other skills. See Appendix A.3 for invocation examples and metadata format.

Skill Extensibility. The `skill-maker` meta-skill enables creation of new domain-specific skills following established patterns. This allows SPOQ deployments to accumulate organizational knowledge as reusable agent capabilities.

5.4 Integration: From Specification to Execution

The implementation components integrate as follows:

1. **Epic Creation:** Human uses `/epic-planning` to decompose a goal. Output: `EPIC.md` + task YAML files in `spoq/epics/active/`.
2. **Validation:** `/epic-validation` and `/task-validation` score specifications against metrics. Failed validations trigger revision.
3. **Execution:** `/agent-execution` reads task files, computes waves, and dispatches Opus workers. Each agent receives task YAML as context.
4. **Tracking:** Agents write journal entries on session completion. The journal accumulates work history for the epic.
5. **Code Validation:** `/agent-validation` scores completed tasks. Failed tasks queue for rework with remediation guidance.
6. **Completion:** On all tasks passing, the epic moves to `spoq/epics/complete/`.

This lifecycle ensures traceability from initial goal through validated delivery, with quality gates preventing propagation of defects. Appendix A.4 discusses integration patterns for CI/CD pipelines, project management tools, and git workflows.

Repository Bootstrap. To reduce adoption friction, SPOQ provides cross-platform installer scripts (`spoq-init.sh` for Linux and macOS, `spoq-init.ps1` for Windows) that automate repository initialization. In fresh-install mode, the scripts create the five-category directory structure (`code/`, `documents/`, `spoq/`, `infrastructure/`, `tests/`), copy the full skill definitions described in Section 5.3, and generate starter `CLAUDE.md` and `journal.md` files. An optional `--full` flag provisions an example epic so that new adopters can exercise the pipeline immediately.

For repositories that already use an earlier directory layout, the `--upgrade` flag activates a migration path. The upgrade routine detects legacy structures (e.g., `automation/tasks/`), relocates epics to `spoq/epics/active/` with conflict-aware merging, refreshes skill definitions from the canonical source while preserving timestamped backups, and updates configuration references. Together, these modes allow a team to adopt or migrate to SPOQ with a single command invocation rather than manual directory scaffolding.

6 Proof-of-Concept Evaluation

We evaluate SPOQ through two detailed case studies on software engineering epics (one internal and one external), followed by a multi-project adoption survey across six repositories spanning distinct technology stacks and problem domains.

Beyond these controlled studies, SPOQ has demonstrated practical value in production settings. The primary author uses SPOQ as the default development workflow across all active projects, finding that the upfront planning investment consistently reduces total delivery time through parallel execution. Ross Sylvester, CEO of Adrata, independently adopted SPOQ as his exclusive development methodology for Adrata's engineering work. He reports that epic planning has become a valuable ideation tool in its own right: decomposing a goal into tasks with dependencies and wave assignments clarifies scope and feasibility before any code is written. Combined with the roadmap

system, this enables him to capture ideas as structured epics and execute them on his own schedule, decoupling planning from implementation.

6.1 Case Study Summaries

We conducted two detailed case studies, one internal and one on an external client codebase. Full wave structures, execution metrics, failure analyses, and lessons learned are provided in Appendix A.5.

Case Study 1: UI Improvements (Internal). This epic modernized a monitoring dashboard with 13 tasks across 2 waves. Wave 0 dispatched 12 independent component tasks in parallel, achieving a 5.3× speedup over sequential execution (3.5 hours vs. 18.5 hours estimated). One agent entered a runaway retry loop during dependency installation, leading SPOQ to adopt a 3-retry maximum with pre-installation verification. Two tasks required rework cycles, yielding a 92% first-pass completion rate.

Case Study 2: Client Website Rebrand (External). This epic rebranded an external B2B sales website across 12 tasks in 4 waves, executed on a codebase maintained by a separate engineering team. The deeper dependency chain limited maximum parallelism to 5 concurrent agents, producing a 2.8× speedup (6.5 hours vs. 18 hours estimated). All 12 tasks completed successfully with 174 passing tests and zero code defects. The primary challenge was test fixture synchronization: three orchestrator interventions were needed when parallel agents' code changes invalidated sibling test assertions. SPOQ now recommends treating test files as implicit dependents of the components they exercise.

6.2 Multi-Project Adoption Survey

Overview. Beyond the two detailed case studies above, SPOQ has been deployed across multiple repositories by two practitioners, spanning distinct technology stacks and problem domains. Table 5 summarizes each completed deployment.

Table 5: SPOQ Adoption Across Completed Projects

Project	Domain	Tasks	Tests	Stack
Savvy Expat	E-commerce	10	154	Next.js, Docker
Railroad OS	Linux tooling	43	55	Bash, i3 WM
SPOQ Website	Documentation	23	18	Next.js, Terraform
Pinpoint Platform	Backend API	16	308	Spring Boot, Java
Pinpoint Infra	Cloud infra	17	—	Terraform, AWS
Pinpoint Analytics	Tracking	7	—	Next.js, GA4
Pinpoint Billing	Payments	6	—	Spring Boot, Stripe

Domain Diversity. The deployments span frontend, backend, infrastructure, and DevOps domains. Savvy Expat rebuilt a relocation services website producing 154 tests across 14 suites. Railroad OS applied SPOQ to Linux window manager configuration (43 tasks), a domain far removed from web development. The Pinpoint ecosystem demonstrates breadth within a single product: a Spring Boot API (16 tasks), AWS infrastructure via Terraform (17 tasks), GA4 analytics with GDPR consent (7 tasks), and Stripe tiered billing (6 tasks).

Aggregate Metrics. Across all completed epics (122 tasks total), average agent confidence scores ranged from 0.90 to 0.95, and all deployments achieved 100% task completion rates. The most common failure mode was test fixture synchronization during parallel execution, observed independently in three separate projects, indicating a systematic challenge rather than a project-specific anomaly.

Execution Velocity. As a concrete demonstration of throughput, the Pinpoint Rebrand epic (Case Study 2, 12 tasks) and a companion analytics epic (7 tasks) were both planned and executed in a single three-hour session using six concurrent Claude Code instances under a single Max license. The 19 combined tasks (spanning content rewriting, persona page creation, navigation updates, analytics instrumentation, and test expansion) were completed from cold start to full verification between 4:00 AM and 7:00 AM, yielding a sustained rate of approximately 6 tasks per hour.

6.3 Failure Modes and Mitigations

Operational deployment of multi-agent systems introduces failure patterns distinct from single-agent development. Through our case studies and deployments, we identified nine categories of operational risk spanning resource contention, context window exhaustion, agent behavioral failures (including runaway loops and validation gaming), coordination conflicts, cost overruns, and security concerns. Table A.11 in the Appendix consolidates these risks with detection signals and mitigations. Appendix A.6 provides detailed analysis of each failure category with specific examples from our deployments.

6.4 Quantitative Summary

Across both case studies and the adoption survey (see Table A.12 in the Appendix for full metrics), SPOQ achieved speedup factors ranging from 1.3× to 5.3× depending on dependency structure. The UI epic achieved the highest speedup due to its embarrassingly parallel Wave 0, while the Rebrand epic's deeper dependency chain limited parallelism but demonstrated strong test coverage growth (134 → 174 tests) and zero code defects despite five concurrent agents. Across the adoption survey, the recurring challenge was test fixture synchronization during parallel execution, which manifested independently in three separate projects.

6.5 Cost Analysis

Under per-token API pricing, a typical Opus worker task costs approximately $1.95 (25K input, 5K output tokens), yielding roughly $28 per 13-task epic. Under Anthropic's flat-rate Max plan ($200/month), effective per-task costs drop to approximately $0.10 at scale, representing a 20× reduction. SPOQ's three-tier hierarchy serves as an economic optimization: reserving Opus tokens for task execution while routing validation and triage through Sonnet and Haiku preserves the most expensive budget for work that demands it. Appendix A.8 provides detailed pricing models, the Director Model scaling paradigm, and ROI framework analysis.

7 Discussion

7.1 Comparison with Prior Multi-Agent Systems

Table 6 contrasts SPOQ with representative multi-agent systems and autonomous coding tools.

Table 6: Comparison of Multi-Agent and Autonomous Coding Approaches

Feature	**SPOQ**	**ChatDev**	**MetaGPT**	**AutoGPT**	**Devin**	**Aider**
Execution model	Wave-parallel	Sequential	Sequential	Recursive	Single-agent	Single-agent
Explicit dependencies	DAG	Implicit	Implicit	Priority queue	None	None
Planning validation	10 metrics	None	Informal	None	None	None
Code validation	10 metrics	Test-based	Review	Self-eval	Test-based	Test-based
Human integration	HaaA	Observer	Observer	Minimal	Minimal	Collaborative
Agent specialization	3-tier	Role-based	Role-based	General	General	General
Parallelism potential	High	Low	Low	Low	None	None

Execution Model. ChatDev and MetaGPT use sequential role-playing where agents take turns. MAD uses synchronous debate rounds. SPOQ achieves true parallelism through wave-based dispatch, with agents in the same wave executing concurrently.

Dependency Management. Prior systems embed dependencies implicitly in role sequences (programmer after designer) or artifact flows. SPOQ makes dependencies explicit in a DAG, enabling static analysis, critical path computation, and parallelism optimization.

Validation Rigor. ChatDev relies on test execution; MetaGPT uses informal review. SPOQ applies structured metrics at both planning and code stages, with quantified thresholds that prevent low-quality work from proceeding.

Human Role. In ChatDev and MetaGPT, humans observe outputs but do not participate in the orchestration loop. SPOQ's HaaA model integrates human judgment at key decision points while preserving automation benefits.

7.2 Limitations

SPOQ has several limitations that future work should address:

Upfront Planning Investment. SPOQ requires detailed epic specifications before execution begins. For exploratory or rapidly-changing requirements, this upfront cost may be prohibitive. Projects with unclear scope may benefit from more adaptive approaches.

Dependency on Human Quality. The HaaA model's effectiveness depends on the human specialist's skill in task decomposition and validation. A human who approves poorly-structured epics will see cascading quality issues.

Empirical Scale. Our evaluation spans nine deployments with 147 completed tasks across diverse domains. While this breadth strengthens generalizability claims relative to a single case study, all deployments share a single primary author. Independent replication by other teams and controlled studies with baseline comparisons are needed to validate speedup and quality claims rigorously.

Reference Implementation Coupling. Our current implementation is coupled to Claude Code and Anthropic's model family, representing a specific instantiation of the SPOQ methodology rather than its only possible form. However, the core methodology, including wave computation, validation scoring, dependency resolution, and journal tracking, is expressed in platform-agnostic YAML and Markdown formats. Porting SPOQ to alternative platforms (Gemini CLI, OpenAI Assistants API, open-source model deployments) would require remapping the capability tiers to the target provider's model offerings and adapting the orchestration interface. The fundamental algorithms and quality gates remain unchanged across implementations.

No Cross-Epic Learning. Each epic starts fresh. SPOQ does not currently transfer lessons from prior epics (successful patterns, common failure modes) to new planning sessions.

Metric Reliability. The 20 validation metrics rely on LLM-based assessment without inter-rater reliability testing. Automated quality scores may exhibit biases, inconsistency across runs, or susceptibility to gaming. Human validation studies comparing LLM scores to expert assessments are needed.

Cost Approximations. Our cost analysis uses estimated token counts that vary significantly by task complexity, codebase size, and context requirements. Real-world costs may differ substantially. The analysis does not include infrastructure, human oversight time, or rework loop costs comprehensively.

7.3 Future Work

Two complementary developments address SPOQ's primary limitations and define the trajectory of the methodology.

LCARS: Solving the Specification Bottleneck. SPOQ's pipeline begins with a human-authored epic specification that serves as the sole source of system context available to executing agents. When this specification is incomplete (a missed dependency, an unmentioned service interaction), agents execute faithfully against an inaccurate model of reality. The quality gates catch *execution* failures but are structurally blind to *specification* failures.

LCARS (Layered Codebase Analysis and Retrieval System) addresses this gap by constructing a continuously-maintained code knowledge graph: a directed, labeled multigraph where vertices represent system entities (services, endpoints, database objects, UI components, infrastructure resources) and edges encode typed relationships (calls, reads, writes, depends-on, tests, deploys). An indexing pipeline extracts entities from five source categories (static analysis, schema analysis, infrastructure-as-code, dynamic tracing, and test mapping) and maintains the graph incrementally on every commit.

When a human declares a change goal, the orchestrator queries the graph to compute a *blast radius*: the subgraph of all entities reachable within a calibrated traversal depth, filtered by edge-type relevance. This replaces hand-authored `files_to_touch` manifests with structurally-determined context assembly. The human's role shifts from exhaustive specification to goal declaration and plan validation. LCARS does not replace SPOQ's dispatch model. It replaces the *input* to SPOQ's dispatch model. The wave DAG, quality gates, and validation cascade remain unchanged. A forthcoming companion paper will present the full graph ontology, traversal semantics, and integration architecture.

Native Platform Support: Claude Code Agent Teams. SPOQ's current orchestration relies on manually-coordinated Claude Code instances. Claude Code's experimental Agent Teams feature provides native primitives that align directly with SPOQ's dispatch model: `TeamCreate` establishes a coordinated session with a lead and specialized teammates, `TaskCreate`/`TaskUpdate` manage a shared task list with dependency tracking, and `SendMessage` enables direct inter-agent communication. Teams support plan approval workflows where teammates operate in read-only mode until the lead validates their approach, mirroring SPOQ's validation gates.

The mapping is natural: SPOQ's wave computation produces task assignments with explicit dependencies; Agent Teams' task list enforces those dependencies through blocked/unblocked state transitions. SPOQ's three-tier hierarchy (Opus workers, Haiku investigators, Sonnet reviewers) maps to Agent Teams' model selection per teammate. The delegate mode, which restricts the lead to coordination-only tools, formalizes SPOQ's orchestrator role separation.

Integrating SPOQ with Agent Teams would shift the methodology from a set of conventions enforced by skill prompts to a native orchestration protocol. Wave dispatch becomes `TaskCreate` with dependency declarations. Validation gates become `TaskCompleted` hooks that invoke Sonnet reviewers before marking tasks done. The journal system feeds naturally into the shared task list's completion records. Combined with LCARS-generated context, this creates a pipeline where the human declares a goal, the graph computes the blast radius, the orchestrator generates a wave-structured task list, and Agent Teams execute it with built-in coordination, messaging, and quality enforcement.

7.4 Broader Implications

SPOQ represents a step toward *AI-native software engineering*: development processes designed around AI capabilities rather than retrofitting AI into human-centric workflows.

Key implications include:

- **Redefined roles:** Engineers shift from writing code to validating agent outputs, decomposing problems, and making architectural decisions.
- **Quality as constraint:** Explicit quality gates force upfront investment in planning, potentially improving overall software quality.
- **Scalable expertise:** A single human specialist can leverage multiple agents, scaling their expertise across more projects than traditional pair programming allows.

The Director Model. At scale, SPOQ enables a single engineer to direct a digital workforce: 6 Claude Code instances (1 planning copilot + 5 execution overseers) coordinating 50–100 concurrent agents across independent epics, achieving daily throughput of 75–150 tasks with output previously requiring 8–10 engineers.

Applicability. SPOQ's structured approach introduces upfront overhead that pays off for projects with 5+ parallelizable tasks spanning 4+ hours of estimated work. Below this threshold, orchestration overhead typically exceeds execution time savings. Appendix A.9 provides a detailed decision framework with applicability guidance by project type.

8 Conclusion

We introduced SPOQ (Specialist Orchestrated Queuing), a methodology for multi-agent software engineering that addresses coordination, quality, and human oversight challenges in prior approaches.

8.1 Summary of Contributions

Wave-Based Topological Dispatch. SPOQ computes parallel execution waves from task dependency graphs, achieving speedups of 2–5× over sequential execution while respecting precedence constraints.

Dual Validation Gates. By validating both planning quality (10 metrics, 95/90 threshold) and code quality (10 metrics, 95/80 threshold), SPOQ catches issues at the stages where they are cheapest to fix.

Human-as-an-Agent Integration. The HaaA paradigm positions the human specialist as a high-value agent within the orchestration loop, enabling bidirectional collaboration that amplifies both human judgment and agent productivity.

Three-Tier Agent Hierarchy. SPOQ's Opus/Sonnet/Haiku hierarchy optimizes cost-quality tradeoffs by matching agent capabilities to role requirements: high capability for execution, balanced for review, fast-cheap for triage.

Practical Lessons. Through proof-of-concept evaluation, we identified and addressed failure modes including runaway retry loops, lock file contention, and context window exhaustion.

8.2 Vision for AI-Native Engineering

SPOQ represents early steps toward a future where engineers become architects and validators rather than line-by-line implementers, where quality is built into process through structured gates, and where human expertise scales through orchestration. Much work remains to realize this vision fully. We hope SPOQ provides a useful framework for researchers and practitioners exploring the frontier of AI-assisted software development.

Acknowledgments

We thank the Claude Code development team for the tooling infrastructure that enabled this research. We also thank Ross Sylvester (CEO, Adrata) and John Armbruster (Founding Engineer, Notary Everyday), whose adoption of SPOQ and candid feedback helped refine the methodology.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Edward G Coffman Jr and Ronald L Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.

Cognition Labs. Devin: The first ai software engineer. https://www.cognition-labs.com/introducing-devin, 2024. Accessed: 2025.

Paul Gauthier. Aider: Ai pair programming in your terminal. https://github.com/paul-gauthier/aider, 2024. Accessed: 2025.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

James E Kelley Jr and Morgan R Walker. Critical-path planning and scheduling. *Proceedings of the Eastern Joint Computer Conference*, pages 160–173, 1959.

LangChain, Inc. LangGraph: Building language agents as graphs. https://github.com/langchain-ai/langgraph, 2024. Accessed: 2025-02-01.

Chen Li, Chen Qian, Xin Cong, Cheng Yang, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. Experiential co-learning of software-developing agents. *arXiv preprint arXiv:2312.17025*, 2023.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Zhaopeng Tu, and Shuming Shi. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118*, 2023.

João Moura. CrewAI: Framework for orchestrating role-playing AI agents. https://github.com/crewAIInc/crewAI, 2024. Accessed: 2025-02-01.

Yohei Nakajima. Babyagi: Ai-powered task management system. https://github.com/yoheinakajima/babyagi, 2023. Accessed: 2025.

Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. Shop2: An htn planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

Anton Osika. Gpt-engineer: Specify what you want it to build, the ai asks for clarification, and then builds it. https://github.com/gpt-engineer-org/gpt-engineer, 2023. Accessed: 2025.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

Toran Bruce Richards. Auto-gpt: An autonomous gpt-4 experiment. https://github.com/Significant-Gravitas/Auto-GPT, 2023. Accessed: 2025.

Xingyao Wang, Boxuan Chen, Ziyi Adler, Yufan Song, Neil Graham, Huazhe Yuan, Shunyu Yao, and Sida Wang. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

A Supplementary Material

This appendix provides extended discussion, detailed examples, and supporting analysis referenced from the main text. Sections are ordered to follow the main paper's structure.

A.1 Threshold Design Rationale

The 95/90 (planning) and 95/80 (code) thresholds were chosen based on practical experience:

- **95% aggregate** ensures overall high quality while allowing minor imperfections in individual metrics.
- **90% per-metric for planning** reflects that planning errors propagate downstream. A plan with 70% dependency graph quality will cause execution failures regardless of other metrics.
- **80% per-metric for code** acknowledges legitimate tradeoffs. A task might score 75% on SOLID adherence for pragmatic reasons while still being acceptable.
- **Plans are cheap to fix.** Re-decomposing tasks costs human time but no agent compute. Code rework costs both.

A.2 Complete Metric Rubrics

This subsection provides detailed scoring rubrics for both validation gates.

A.2.1 Planning Validation Rubrics

Vision Clarity (VC).

- 100: Clear overview with problem statement, solution, scope boundaries, and end state
- 80: Clear overview, minor ambiguity in scope
- 60: Overview present but vague
- 40: Overview missing key context
- 0: No overview or incomprehensible goal

Architecture Quality (AQ).

- 100: ASCII/visual diagram with all components, relationships, and data flow
- 80: Diagram present, minor gaps in explanation
- 60: Text-only architecture, no diagram
- 40: Partial architecture description
- 0: No architecture section

Task Decomposition (TD).

- 100: Atomic tasks (1–4h), independent where possible, complete coverage
- 80: Good decomposition, minor overlap
- 60: Some tasks too large or overlapping
- 40: Significant decomposition issues
- 0: Tasks not properly decomposed

Dependency Graph (DG).

- 100: Visual graph, all dependencies valid, no cycles, optimal ordering
- 80: Graph present, dependencies valid, minor ordering improvements possible
- 60: Text-only dependencies, all valid
- 40: Some invalid dependencies or missing graph
- 0: Circular dependencies or critically broken graph

Coverage Completeness (CC). Score equals percentage of success criteria mapped to tasks: $\text{CC} = 100 \times \frac{|\text{covered criteria}|}{|\text{total criteria}|}$

Phase Ordering (PO).

- 100: Phases follow logical progression, maximum parallelism exploited
- 90: Good ordering, minor parallelism opportunities missed
- 75: Ordering works but inefficient
- 60: Some ordering violations
- 0: Critical ordering errors (dependent tasks in same wave)

Scope Coherence (SC).

- 100: All tasks directly serve epic goal
- 80: 1 tangential task
- 60: 2–3 tangential tasks
- 40: Multiple unrelated tasks
- 0: Tasks don't align with epic goal

Success Criteria Quality (SQ).

- 100: All criteria SMART, checkbox format, testable
- 90: Criteria measurable, minor gaps
- 75: Some criteria vague
- 60: Multiple unmeasurable criteria
- 0: No success criteria or all vague

Risk Identification (RI).

- 100: Risks section with likelihood, impact, and mitigations
- 90: Risks mentioned, implicit mitigations
- 75: Some risks noted, no mitigations
- 60: Risks not addressed
- 0: Critical risks ignored

Integration Strategy (IS).

- 100: Clear integration points, verification steps between phases
- 90: Integration implicit but clear
- 75: Some integration points unclear
- 60: Integration strategy missing
- 0: Tasks cannot be integrated as designed

A.2.2 Code Validation Rubrics

Syntactic Correctness (SC).

- 100: Compiles cleanly, 0 warnings
- 80: Compiles with minor warnings
- 60: Compiles with significant warnings
- 0: Does not compile

Test Existence (TE). Score equals percentage of new public methods with corresponding tests: $\text{TE} = 100 \times \frac{|\text{tested methods}|}{|\text{new public methods}|}$

Test Pass Rate (TP). Score equals percentage of tests passing: $\text{TP} = 100 \times \frac{|\text{passing tests}|}{|\text{total tests}|}$

Requirements Fidelity (RF).

- 100: All requirements fully implemented
- 80: All core requirements met, minor gaps
- 60: Core requirements met, some missing
- 40: Partial implementation
- 0: Does not address requirements

SOLID Adherence (SA). 20 points per principle (S, O, L, I, D) based on degree of adherence.

Security (SE). Start at 100, deduct for OWASP Top 10 vulnerabilities: SQL injection (-100), command injection (-100), XSS (-60), broken auth (-80), sensitive data exposure (-60), etc.

Error Handling (EH). 20 points each for: I/O wrapped in try-catch, meaningful error messages, proper logging, resource cleanup, safe user-facing errors.

Scalability (SL). Based on algorithm complexity of hot paths: O(1)/O(log n)/O(n) = 100, O(n log n) = 90, $O(n^2) = 40$, $O(2^n) = 0$.

Code Clarity (CC).

- 100: Crystal clear, reads like well-written prose
- 80: Clear with minor naming improvements possible
- 60: Understandable but requires effort
- 40: Confusing structure or naming
- 0: Unreadable, magic numbers, cryptic names

Completeness (CO).

- 100: Complete, no loose ends
- 80: Minor polish needed
- 60: Some TODOs remain but core is done
- 40: Significant unfinished sections
- 0: Stub implementations, placeholders

Automatic deductions: TODO (-25), FIXME (-25), NotImplementedException (-30).

A.3 Implementation Details

This subsection provides complete schema examples, journal entry formatting, and skill metadata referenced from Section 5.

A.3.1 Epic Directory Structure

Each epic occupies its own directory under `spoq/epics/`:

Listing 1: Epic directory layout

```
spoq/epics/
  active/                    # Epics in progress
    epic-name/
      EPIC.md
      tasks/
        01-init-project.yml
        ...
  complete/                  # Finished epics
  ROADMAP.md                 # Priority tracker
```

The `EPIC.md` file provides context: goal statement, architecture diagrams, success criteria, dependency visualization, wave assignments, effort estimates, and risk assessment. Individual task files contain execution-ready specifications.

A.3.2 Task YAML Schema Examples

The task specification (Definition 5.1) organizes fields into three categories. The following listings illustrate each category with a representative task.

Identity Fields. These establish task context within an epic:

Listing 2: Task identity fields

```
id: 04-content-constants       # Unique within epic
title: Create Content Constants File
epic: spoq-website             # Parent epic reference
```

Execution Control Fields. These govern scheduling, effort estimation, and dependency resolution:

Listing 3: Execution control fields

```
status: pending                # pending | in_progress | completed
priority: high                 # critical | high | medium | low
phase: 1                       # Wave assignment (0 = no deps)

estimate:                      # PERT three-point estimate
  optimistic: 15m
  realistic: 45m
  pessimistic: 2h

dependencies:                  # Task IDs that must complete first
  - 01-init-project
  - 02-setup-deps

skills_required:               # Domain knowledge needed
  - typescript
  - react
```

Deliverable and Verification Fields. These define expected outputs and acceptance criteria:

Listing 4: Deliverable and verification fields

```
files_to_touch:                # All files to be modified
  - src/lib/constants.ts
  - tests/constants.test.ts

outputs:                       # Tangible deliverables
  - "Constants file with typed exports"
  - "Unit tests achieving 100% coverage"

acceptance_criteria:           # Verification checklist
  - "[ ] TypeScript compiles without errors"
  - "[ ] All tests pass: `npm test constants`"
  - "[ ] No hardcoded strings in component files"
```

Description Field. The description provides structured implementation guidance using Markdown:

Listing 5: Task description structure

```
description: |
  ## Objective
  Create a centralized constants file for UI strings.

  ## Steps
  1. Create src/lib/constants.ts with typed exports
     ```typescript
     export const SITE_NAME = "SPOQ" as const;
     ```
  2. Add unit tests in tests/constants.test.ts
  3. Replace hardcoded strings in existing components

  ## Verification
  ```bash
  npm run typecheck && npm test constants
  ```
```

A.3.3 Journal Entry Format

Journal entries (Definition 5.2) use YAML frontmatter followed by Markdown sections. The frontmatter captures machine-readable metadata:

Listing 6: Journal entry frontmatter

```
---
agent: Claude Code (Opus 4.5)
start_time: 2025-11-01T10:52:34Z  # ISO 8601 UTC
end_time: 2025-11-01T11:23:15Z
confidence: 0.92                  # 0.0-1.0 calibration score
session_type: development         # development | refactor | bugfix | ...
files_modified:
  - src/components/Hero.tsx
  - tests/hero.test.tsx
tasks_completed: 2
tasks_total: 3
---
```

The body follows a standardized section layout for consistent parsing:

Listing 7: Journal entry body structure

```
## Summary
Brief 1-2 sentence overview of accomplishments.

## Work Completed
- Task 09: Hero section component
- Task 10: Features grid with responsive layout

## Changes Made
**Frontend Components**
- `Hero.tsx` - Implemented animated gradient background
- `Features.tsx` - Added 6-card responsive grid

## Issues Encountered
None (or specific issues with resolutions)

## Testing
- Unit tests: PASS (12/12)
- Integration tests: PASS (4/4)

## Next Steps
1. Implement call-to-action section
2. Add accessibility attributes
```

Tooling Support. SPOQ provides utility scripts for journal management:

- `get-time.py`: Captures accurate UTC timestamps
- `archive-journal.py`: Auto-archives when exceeding 1500 lines
- `parse-to-db.py`: Extracts entries into SQLite for analysis
- `create-mega-journal.py`: Consolidates archives for reporting

A.3.4 Confidence Score Interpretation

Table A.7: Confidence Score Interpretation

Score Range	Interpretation
0.95–1.0	Thoroughly tested, production-ready
0.85–0.94	Well tested, minor edge cases possible
0.75–0.84	Functional, additional testing recommended
0.65–0.74	Works but requires validation
<0.65	Experimental or known issues present

A.3.5 Skill Inventory

Table A.8: SPOQ Skill Inventory

Skill	Stage	Purpose
epic-planning	Planning	Decompose goals into epics with dependency graphs
epic-validation	Validation	Score epics against 10 planning metrics
task-validation	Validation	Score individual tasks before execution
agent-execution	Execution	Orchestrate parallel agent swarms
agent-validation	Validation	Score completed work against 10 code metrics
journal-tracker	Cross-cutting	Track sessions with confidence scores

A.3.6 Skill Metadata and Invocation

Skills are invoked via slash commands that expand into full prompts:

Listing 8: Skill invocation examples

```
# Plan a new epic
/epic-planning "Implement␣user␣authentication␣system"

# Validate before execution
/epic-validation @spoq/epics/active/auth-system

# Execute with parallel agents
/agent-execution @spoq/epics/active/auth-system

# Validate completed work
/agent-validation
```

Each skill's `SKILL.md` includes YAML frontmatter specifying metadata:

Listing 9: Skill metadata format

```
---
name: epic-planning
description: Decompose high-level goals into structured epics
             with atomic tasks and dependency DAGs.
---
```

The body provides:

- **When to Use:** Activation criteria
- **Core Patterns:** Documented approaches with examples
- **Quality Criteria:** Verification standards
- **Integration Points:** Connections to other skills

A.4 Integration Considerations

A recurring question from practitioners concerns SPOQ's integration with existing development infrastructure. We address this honestly: SPOQ currently operates as a standalone orchestration layer invoked via slash commands, without native integrations with external tools. However, its design accommodates several integration patterns.

CI/CD Pipeline Patterns. SPOQ's wave-based execution maps naturally to CI/CD pipeline stages. Wave 0 corresponds to setup and dependency installation, middle waves to parallel build and test jobs, and final waves to integration testing and deployment. In our infrastructure case study, the GitLab CI pipeline was updated as a task within the SPOQ epic itself, demonstrating that CI/CD configuration is orchestrable work rather than requiring special integration.

For automated triggering, organizations could invoke SPOQ from pipeline scripts when epic YAML files change, treating the orchestrator as a build step. A GitLab CI job might monitor the `spoq/epics/` directory and trigger agent execution on detected changes. This pattern remains unexplored in our evaluation but represents a natural extension. GitHub Actions workflows could similarly invoke SPOQ as a step, potentially enabling fully automated epic execution on pull request events.

Project Management Alignment. SPOQ task YAML files parallel the structure of Jira stories and Linear issues: both contain titles, descriptions, acceptance criteria, and estimates. The structural similarity suggests bidirectional sync is feasible; a task defined in SPOQ could create a corresponding Jira ticket, and vice versa. However, SPOQ does not currently implement such synchronization, which may create duplicate tracking overhead for teams already committed to existing PM tools.

The journal system partially addresses visibility by providing an audit trail of work completed, though it is designed for agent coordination rather than project management. Teams could export journal entries to their PM tools as a manual bridge, accepting the overhead until native integration warrants development investment.

Git Workflow Recommendations. Parallel agent execution raises questions about branch strategy. Our current approach uses a single working branch with agents operating on disjoint files, ensured by task `files_to_touch` specifications. This works well for up to 12 concurrent agents but may encounter contention at larger scales.

For larger teams or epics with overlapping file modifications, we recommend git subtrees for epic isolation: each epic operates in its own subtree, with automated merges after validation gates pass. Feature branches per wave provide an alternative, with wave N's branch merging into main before wave N+1 begins. Both approaches add orchestration complexity but reduce merge conflicts.

IDE Integration Possibilities. SPOQ's slash-command interface suggests IDE integration opportunities. A VS Code extension could display the current epic's dependency graph, highlight files assigned to each task, and show validation scores in real time. JetBrains plugins could integrate journal entries into the project view, surfacing agent work history alongside version control logs.

Such integrations remain future work. Our current implementation prioritizes methodology validation over tooling polish, reflecting a conscious choice to prove the approach before investing in developer experience enhancements.

Future Integration Directions. Several integration directions merit exploration: (1) native Jira/Linear/Asana adapters for bidirectional task sync; (2) Slack/Teams webhooks for wave completion notifications; (3) GitHub/GitLab API integration for automated PR creation per wave; (4) IDE extensions for real-time epic visualization. These would reduce friction for teams adopting SPOQ within established toolchains, though each adds maintenance burden. We advocate starting with SPOQ as a standalone layer, adding integrations only when adoption justifies the investment.

A.5 Detailed Case Studies

A.5.1 Case Study 1: UI Improvements Epic

Epic Overview. The UI improvements epic modernized a monitoring dashboard with toast notifications, data tables, and API key management components. The epic comprised 13 tasks across 2 waves.

Wave Structure.

- **Wave 0 (12 tasks):** Independent component implementations including Sonner setup, DataTable component, modal dialogs, toast integrations, search functionality, and tests.
- **Wave 1 (1 task):** End-to-end QA depending on all Wave 0 tasks.

Execution Metrics. Table A.9 summarizes the execution.

Table A.9: UI Improvements Epic Execution

Metric	Value
Total tasks	13
Wave 0 parallelism	12 concurrent agents
Wave 1 parallelism	1 agent
Sequential estimate	18.5 hours
Parallel estimate	3.5 hours
Speedup factor	5.3×
Tasks completed	12 of 13 (92%)
Rework cycles	2 (tasks 04, 07)

Failure Mode Encountered. Task 04 entered a runaway retry loop, executing `npm install sonner` over 100 times. The agent failed to recognize that the dependency was already installed and continued retrying indefinitely.

Lesson Learned. SPOQ now enforces a maximum of 3 retries per installation command and requires agents to verify existing dependencies before installation attempts.

A.5.2 Case Study 2: Client Website Rebrand

Epic Overview. This epic rebranded an external client's sales website (Pinpoint, a B2B QA testing platform) from founder-centric messaging to developer-focused positioning. The work included removing biographical content, rewriting homepage copy, creating three persona-specific landing

pages, updating navigation and SEO metadata, and expanding the test suite. The epic comprised 12 tasks across 4 waves and was executed on a codebase maintained by a separate engineering team.

Wave Structure.

- **Wave 0 (2 tasks):** Content removal and routing scaffold (parallel).
- **Wave 1 (5 tasks):** Homepage rewrite, pricing update, and three persona landing pages (parallel).
- **Wave 2 (3 tasks):** Navigation, SEO, and section refinement (parallel).
- **Wave 3 (2 tasks):** Test expansion and accessibility polish (parallel).

Execution Metrics. Table A.10 summarizes the execution.

Table A.10: Client Rebrand Epic Execution

Metric	Value
Total tasks	12
Max parallelism	5 concurrent agents (Wave 1)
Sequential estimate	18 hours
Parallel estimate	6.5 hours
Speedup factor	2.8×
Tasks completed	12 of 12 (100%)
Orchestrator interventions	3
Test suites / cases	20 / 174 (0 failures)

Test Fixture Synchronization. All three orchestrator interventions involved test assertions falling out of sync with code changes made by parallel agents. In Wave 1, a pricing component gained new required props (`price`, `period`) that the existing accessibility test did not supply. In Wave 2, updated navigation links and SEO strings invalidated tab-order and metadata assertions. None of these were code defects; they were coordination gaps where one agent's changes invalidated another agent's test fixtures.

Lesson Learned. Structural component changes must propagate to all dependent test fixtures within the same wave. SPOQ now recommends treating test files as implicit dependents of the components they exercise, ensuring fixture updates are included in the same task that modifies the component interface.

A.6 Failure Modes and Mitigations (Detail)

The following subsections provide detailed analysis of each failure category identified during SPOQ development and deployment, with specific examples and comprehensive mitigation strategies.

A.6.1 Resource Contention Failures

Lock File Contention. Multiple agents running `npm install` concurrently cause lock file conflicts, manifesting as `EBUSY` errors or corrupted `node_modules` directories. In one deployment, three parallel agents attempted dependency installation simultaneously, resulting in a 12-minute debugging session.

Mitigation: SPOQ designates a single Wave 0 task for dependency installation, with subsequent tasks assuming dependencies are available. Organizations should extend this pattern to any shared mutable resource: database migrations, cache warming, and artifact generation.

Build Directory Conflicts. Concurrent build processes can corrupt shared directories (e.g., `.next/`, `dist/`, `target/`). Symptoms include partial builds, missing assets, and non-deterministic test failures.

Mitigation: Build verification runs sequentially between waves, not during waves. For CI/CD integration, consider isolated build directories per agent or containerized build environments.

A.6.2 Context and Memory Failures

Context Window Exhaustion. Complex tasks with extensive codebase context can exhaust LLM context windows (currently 200K tokens for Claude). Symptoms include: agents losing track of earlier requirements, producing incomplete solutions, or generating responses that contradict prior instructions. In one knowledge-graph pipeline deployment, an agent with 180K tokens of context began hallucinating function signatures that did not exist in the codebase.

Mitigation: Task descriptions should be self-contained, and the `files_to_touch` specification should limit scope to files the agent genuinely needs. For large codebases, consider:

- Task-specific context manifests listing only relevant files
- Summarization of peripheral code rather than full inclusion
- Breaking complex tasks into sub-tasks with narrower scope
- Context budget monitoring with alerts at 70% utilization

Context Window Management. Long QA feedback exhausted agent context windows. The 20-line remediation limit was introduced to ensure feedback remains actionable without consuming excessive context. Reviewers now prioritize critical issues and defer minor suggestions to follow-up tasks.

A.6.3 Agent Behavioral Failures

Runaway Detection. Agents sometimes enter infinite loops on transient errors, repeatedly executing the same failing command. In Case Study 1, Task 04 entered a runaway retry loop, executing `npm install sonner` over 100 times before intervention.

Mitigation: SPOQ monitors for repeated identical commands (5+ occurrences) and halts with a human consultation request. Organizations should implement:

- Command deduplication with exponential backoff
- Per-task execution time limits (default: 30 minutes)
- Anomaly detection for unusual command patterns

Validation Gaming. Agents optimizing for validation metrics may produce code that passes automated checks while failing to address underlying requirements. Examples include: tests that assert `true === true`, implementations that short-circuit edge cases, and documentation that restates function signatures without explaining behavior.

Mitigation: The dual validation framework addresses surface-level gaming through complementary metrics (test existence vs. test pass rate, syntactic correctness vs. requirements fidelity). However, sophisticated gaming requires human oversight:

- Periodic human code review of high-complexity tasks
- Mutation testing to verify test quality
- Requirements traceability audits
- Cross-validation between independent agents on critical paths

Dependency Resolution Failures. Agents may specify incorrect dependency versions, introduce circular dependencies, or fail to recognize version conflicts. These failures often manifest only at runtime or during integration testing.

Mitigation:

- Lock files (`package-lock.json`, `Cargo.lock`) committed after Wave 0 dependency installation
- Dependency audit as part of code validation metrics
- Version pinning policies enforced via task templates
- Integration tests run at wave boundaries to detect compatibility issues

A.6.4 Coordination Failures

Agent Coordination Failures. In parallel execution, agents may make conflicting assumptions about shared state. Example scenarios include:

- Two agents independently creating the same utility function with different signatures
- An agent assuming a database table exists while another agent is still creating it
- Conflicting CSS class names or component identifiers

Mitigation: Wave boundaries serve as synchronization points. Additional measures include:

- Explicit interface contracts defined in task prerequisites
- Naming conventions specified in task templates
- Linting rules to detect common conflict patterns
- Post-wave merge verification before proceeding

Human Consultation Latency. When agents request human consultation (HaaA), execution pauses until response. If the human is unavailable, this creates a bottleneck that can stall entire wave progressions.

Mitigation:

- Configurable timeout with fallback behavior (skip, abort, or proceed with best-effort)
- Escalation to secondary human reviewers
- Asynchronous consultation queues for non-blocking requests
- Clear documentation of expected response times per request priority

A.6.5 Cost and Resource Failures

Cost Runaway Risk. Without limits, agents retrying failed tasks or entering verbose output loops can rapidly consume API credits. A single runaway agent in Case Study 1 consumed approximately $15 in tokens before detection. Extrapolated to parallel execution with 10+ agents, uncontrolled failures could exceed $100/hour.

Mitigation:

- Per-task token budgets (recommended: 50K input, 10K output for standard tasks)
- Per-epic cost caps with automatic suspension
- Real-time cost monitoring dashboards with alerting
- Graduated rate limiting: warning at 80%, throttle at 90%, halt at 100%

A.6.6 Security and Isolation Failures

Execution Isolation. Agents execute commands in the host environment without sandboxing. Malformed or malicious code could damage the system, exfiltrate data, or consume resources. While LLM agents are unlikely to be intentionally malicious, hallucinated commands (e.g., `rm -rf /`) pose real risks.

Mitigation:

- Container-based execution (Docker) with ephemeral workspaces
- Read-only filesystem mounts where possible
- Network isolation for tasks that do not require external access
- Resource limits (CPU, memory, disk quotas)
- Command allowlisting for high-risk operations
- Audit logging of all executed commands

Rollback and Recovery. If agent work corrupts the codebase, recovery requires manual git operations. SPOQ does not currently implement automatic rollback, relying instead on git history and manual intervention.

Mitigation:

- Commit checkpoints at wave boundaries, enabling `git revert` to known-good states
- Branch-per-epic isolation with squash merge on completion
- Automated backup before epic execution
- Recovery runbooks documenting rollback procedures for common failures

A.7 Evaluation Tables

Table A.11: Operational Risk Summary

Risk Category	**Detection Signal**	**Mitigation**
Context exhaustion	Output truncation	Scope limits, context budgets
Cost runaway	Token counter spikes	Per-task budgets with alerts
Human bottleneck	Queue timeout	Configurable fallback
Lock contention	EBUSY errors	Single-agent dep. install
Validation gaming	High scores, low quality	Human review, mutation testing
Agent conflicts	Merge failures	Interface contracts, wave sync
Runaway loops	Repeated commands	Deduplication, time limits
Security breach	Unauthorized commands	Container isolation, allowlists
Data corruption	Test/build failures	Wave checkpoints, rollback

Table A.12: Evaluation Summary

Metric	**UI Epic**	**Rebrand Epic**	**Adoption (agg.)**
Tasks	13	12	122
Waves	2	4	varies
Max parallelism	12	5	4
Speedup factor	5.3×	2.8×	1.3–3.0×
Completion rate	92%	100%	100%
Orchestrator interventions	1	3	varies
Rework cycles	2	0	0–1
Test cases	—	174	295
Avg. confidence	—	0.92	0.90–0.95

A.8 Cost Analysis and ROI Framework

A critical consideration for SPOQ adoption is the economic viability of multi-agent orchestration. We analyze costs under two pricing models available for Claude API access as of February 2025.

Pricing Model A: Per-Token API. Table A.13 presents the current Claude API pricing structure and how each tier maps to SPOQ agent roles.

Table A.13: Claude API Pricing by Model Tier

Model	Input	Output	SPOQ Role
Opus 4.6	\$15/M tokens	\$75/M tokens	Worker agents
Sonnet 4.5	\$3/M tokens	\$15/M tokens	Reviewer agents
Haiku 4.5	\$0.25/M tokens	\$1.25/M tokens	Investigator agents

Based on observed token consumption in our case studies, a typical Opus worker task consumes approximately 25,000 input tokens and 5,000 output tokens, yielding a per-task cost of approximately \$1.95. For an epic of 13 tasks (similar to our UI improvements study), the total worker cost reaches approximately \$28, excluding reviewer and investigator overhead.

Pricing Model B: Flat-Rate Max Plan. Anthropic's Max plan (\$200/month) provides 20× the usage allowance of a standard Claude Pro subscription. There is no platform-imposed limit on concurrent Claude Code instances; the practical ceiling is human attention. Crucially, usage is metered in two separate buckets: *Opus* consumption and *non-Opus* consumption (Sonnet, Haiku). This two-bucket structure makes SPOQ's three-tier agent hierarchy an economic optimization as well as a capability one; reserving Opus tokens for task execution while routing validation and triage through Sonnet and Haiku preserves Opus headroom for the work that demands it most.

- **Fixed monthly cost** with predictable budgeting
- **Concurrency:** unlimited instances; bounded only by the two usage buckets and human supervisory bandwidth
- **Tiered metering:** Opus budget for Workers; Sonnet/Haiku budget for Reviewers and Investigators
- **Daily capacity:** 50–100 tasks (assuming 4–6 task cycles across active instances)
- **Effective per-task cost:** \$0.10 at scale (100 tasks/day × 20 working days)

At scale, the Max plan reduces per-task costs by approximately 20× compared to per-token pricing, making aggressive parallelization economically viable on a single subscription.

The Director Model. We propose a scaling paradigm we term the *Director Model*, wherein a single human engineer orchestrates multiple Claude Code instances under one Max license:

- **1 planning instance:** Assists with epic decomposition and dependency analysis
- **5 execution instances:** Each runs SPOQ's wave-based dispatch with parallel sub-agents
- **Daily output:** 50–100 completed tasks

In our experience, six concurrent instances represents the human multitasking limit; the usage buckets are rarely exhausted even at this level, leaving headroom for sustained parallel execution. This configuration enables a single engineer to achieve throughput equivalent to 5–8 traditional engineers, representing a qualitative shift in individual productivity potential.

ROI Framework. Table A.14 presents a monthly ROI calculation for the Director Model at typical utilization.

Table A.14: Director Model Monthly ROI Estimate

Component	**Value**
Monthly tasks completed	2,000 (100/day × 20 days)
Equivalent engineer output	5–8 engineers
Traditional cost (8 engineers @ $12,500/mo)	$100,000
Director Model cost (1 Max license)	$200
Engineer salary (director)	$12,500
Total Director Model cost	$12,700
Monthly savings	$87,300
ROI multiplier	7.9×

Under favorable assumptions (consistent utilization, stable task completion), the Director Model yields approximately 8× cost efficiency compared to traditional staffing.

Caveats and Variability. These estimates carry significant uncertainty:

1. **Task complexity variance:** Our cost-per-task figures derive from 1–4 hour tasks. Complex tasks requiring extended context windows or multiple rework cycles can cost 3–5× the baseline.
2. **Rework overhead:** Failed validations requiring remediation add approximately 40% overhead on affected tasks.
3. **Output quality equivalence:** The ROI comparison assumes agent-completed tasks are comparable in quality to human-authored work. This equivalence has not been empirically validated; agent output may require additional review or refinement that narrows the cost gap.
4. **Human supervision cost:** The Director Model assumes skilled engineers capable of effective agent orchestration. Training and context-switching costs are not included.
5. **API rate limits:** Per-token pricing may encounter rate limits at high parallelism, while Max plan throughput is bounded by the usage bucket allowances under sustained heavy parallelism.

6. **Quality vs. speed tradeoffs:** Higher parallelism may introduce integration issues that offset time savings with debugging overhead.

Despite these caveats, the order-of-magnitude cost advantage suggests that multi-agent orchestration represents an economically viable paradigm for software development at scale.

A.9 When to Use SPOQ: Decision Framework

SPOQ's structured approach introduces upfront overhead (task decomposition, YAML specification, dependency mapping, and validation gates) that pays off for certain project types but hinders others. This subsection provides decision guidance for practitioners.

Appropriate Use Cases. SPOQ adds value when:

- **Structured features:** New functionality with clear requirements spanning multiple components. When you can articulate what needs to be built before starting, SPOQ's planning investment translates into parallel execution benefits.
- **Multi-file refactors:** Coordinated changes across a codebase where parallelism accelerates delivery. Renaming a widely-used interface, migrating to a new library, or restructuring module boundaries benefit from explicit dependency tracking.
- **Infrastructure work:** Terraform configurations, CI/CD pipelines, and deployment automation have well-defined outputs and benefit from explicit dependency specification.
- **Team handoffs:** Projects requiring audit trails, explainability for stakeholders, or handoffs between engineers benefit from SPOQ's journal tracking and task documentation.
- **Scale thresholds:** Our empirical data suggests SPOQ provides net benefit for epics with 5+ tasks spanning 4+ hours of estimated work. Below this threshold, orchestration overhead often exceeds execution time.

Inappropriate Use Cases. SPOQ's overhead exceeds its value when:

- **Exploratory prototyping:** When requirements are unclear and the goal is discovery, SPOQ's upfront planning becomes a burden. Rapid iteration with a single agent or direct coding serves exploration better.
- **Small fixes:** Single-file bug fixes or minor changes where task decomposition time exceeds execution time. A 10-minute fix should not require 30 minutes of specification.
- **Creative flow states:** Sessions where the developer seeks direct engagement with code, what practitioners sometimes call "vibe coding", benefit from immediacy, not delegation. SPOQ assumes you want to orchestrate, not implement.
- **Urgent hotfixes:** Time-critical patches where validation gates delay resolution. When production is down, skip the methodology and fix the problem directly.
- **Learning exercises:** When the goal is understanding a codebase or technology, doing the work yourself provides education that delegation cannot.

Decision Framework. A simple heuristic: if specifying tasks and dependencies would take longer than executing the work directly, SPOQ is likely overkill. Conversely, if you can identify 5+ subtasks with clear deliverables and the total estimated effort exceeds 4 hours, SPOQ's parallelism and quality gates typically justify the planning investment.

Table A.15 summarizes common scenarios:

Table A.15: SPOQ Applicability by Project Type

Project Type	**Use SPOQ?**	**Rationale**
New feature (5+ files)	Yes	Parallelism benefits; quality gates prevent rework
Single-file bug fix	No	Overhead exceeds benefit
Exploratory prototype	No	Requirements unclear; planning premature
Migration/refactor	Yes	Coordination critical; dependency tracking valuable
Learning exercise	No	Flow state preferred; education through doing
Production hotfix	No	Urgency trumps process
Infrastructure epic	Yes	Well-defined outputs; explicit dependencies
Documentation update	Depends	Multi-file: yes; single page: no

Overhead vs. Payoff. SPOQ's overhead includes: (1) epic planning and task decomposition (30–90 minutes), (2) YAML specification (5–10 minutes per task), (3) epic validation iteration (10–30 minutes), and (4) journal tracking and review (ongoing). For a 10-task epic, expect 2–3 hours of orchestration overhead.

The payoff comes from: (1) parallel execution reducing wall-clock time, (2) validation gates catching issues before they cascade, (3) reduced context-switching as agents handle implementation details, and (4) audit trails simplifying review and handoffs. Across our case studies and adoption survey, speedups ranged from 1.3× to 5.3× depending on dependency structure and parallelization potential.

The break-even point varies by developer productivity and agent capability, but our experience suggests epics under 5 tasks rarely justify SPOQ's overhead, while epics over 15 tasks almost always do. The 5–15 task range requires judgment about parallelization potential and team coordination needs.