Intelligent Defect Analysis and Resolution System

By Reginald Rhoe

A Project Submitted to
California Science and Technology University
1601 McCarthy Blvd • CSTU
Milpitas, CA 95035
Phone 408.400.3948 • Fax 510.474.1861

A Partial Fulfillment of the Requirements
For the certification of
Emerging Technology Training Program

12/15/2025

Table of Contents

Chapter 1: Introduction

In recent years, artificial intelligence (AI) has emerged as a transformative force in software development, particularly in the areas of defect detection and resolution. This capstone project explores the development and implementation of an AI-driven system designed to identify and address software defects efficiently. The motivation behind this project stems from the increasing complexity of software systems and the need for automated tools that can enhance the accuracy and speed of defect management. The current focus has been on AI assistance for the individual developer; however, this application provides an integrated approach not just for the individua contributorl, but also serves as an additional tool that assists the oversight management functions of an organization's engineering department. The following chapters outline the methodology, system implementation, results, and future directions for this AI-based solution. The deliverable for this capstone project is a proof of concept and a minimal viable product.

AI-powered code review and analysis system that combines multiple intelligence sources to provide deep insights into the codebase:

What Parts it Has"
-Git Integration: Direct access to commit history, diffs, and file changes
-Qdrant RAG: Semantic search across the indexed codebase
-OpenAI LLM: Advanced language model for code analysis and recommendations
-CrewAI Agents: Specialized agents are created for different analysis task.

How It Works

1.Code Ingestion:: The repository code is indexed into the Qdrant vector database
2.Task Submission: Tasks are created to describe the desired analysis
3.Agent Analysis: Agents retrieve relevant context from Git and Qdrant.
4.Results: Comprehensive analysis with code quality feedback is provided to users.

Chapter 2: Business Market

Current Landscape

The AI-assisted coding market has evolved rapidly, embedding intelligent tools directly into IDEs and platforms like GitHub. Tools such as GitHub Copilot, Amazon CodeWhisperer, and Tabnine provide:

- Context-aware code completion
- Natural language-to-code generation
- Automated documentation

These tools streamline workflows by suggesting entire code blocks, improving debugging, and enabling rapid refactoring.

On GitHub, AI extends beyond coding assistance to include:

- Copilot Chat for conversational help
- Automated pull request summaries
- CLI support
- Security vulnerability fixes through Copilot for Security

Collectively, these innovations accelerate development, enhance code quality, and integrate seamlessly with cloud-based environments—signaling a future where AI-driven collaboration and automation become standard in software engineering.

Next Generation: Intelligent Defect Analysis and Resolution System

Advantage: Complete picture with temporal, semantic, and analytical context.

While current tools provide valuable assistance, they remain limited to single-point interactions. This framework introduces a multi-agent orchestration platform that addresses enterprise-scale coordination and contextual intelligence. While traditional AI coding assistants excel at generating code from current context, this intelligent framework represents a significant architectural evolution by implementing ****multi-source Retrieval-Augmented Generation (RAG)**** that combines temporal, semantic, and analytical intelligence:(Table 1)

Complementary Data Sources Architectural Advantages in Intelligent Framework Context

Question Type	Answered By
"What changed?"	Git Tools
"When was this introduced?"	Git Tools
"Who made this change?"	Git Tools
"How does X work?"	Qdrant + LLM
"Where is similar code?"	Qdrant
"Why was this changed?"	Git (message) + LLM (reasoning)
"What's the impact?"	Git (stats) + Qdrant (related code) + LLM (analysis)

Table 1: IDE Gaps

Capability	IDE Assistants	Intelligent Framework
Context Scope	Current file + open files	Entire codebase + history
Temporal Analysis	No commit history	Full Git history with metadata
Cross-file Analysis	Limited to visible files	Semantic search across all files
Pattern Detection	Current code patterns	Historical patterns + trends
Team Coordination	None	JIRA, GitLab, team assignment
Requirements Tracking	None	CAMEO, requirements alignment
Root Cause Analysis	Surface-level	Deep historical + multi-file
Defect Management	None	Full defect lifecycle + patterns

Chapter 3: Methodology

The methodology for developing the Intelligent Defect Analysis and Resolution System is grounded in an integrated framework designed to enhance software development and quality assurance processes across the entire software development lifecycle. This approach leverages the synergy between a Master Control Panel (MCP) and CrewAI, a robust Python library, to deliver a comprehensive, AI-driven solution for defect management. The framework is designed to interface and augment IDE AI assist and chat tools like Open WebUI. In the MVP, the term MCP refers to an implemented container to represent a **Model Context Protocol,** which would most likely be installed in an enterprise system to handle some of the functions.

System Idea and Rationale

The proposed system is deployed directly within environments such as Gitlab or the VSCode IDE, empowering development teams to launch autonomous tasks triggered by key events—like commits, issue creation, or pipeline test failures. Unlike existing tools that focus solely on defect tracking or predictive analytics, this system seamlessly integrates natural language processing, , and machine learning to improve the full spectrum of defect handling. It analyzes live defect data in real time, identifies systemic issues, streamlines workflows, and generates actionable insights for both individual developers but also the leadership level such as Scum Master, Project management, Quality Assurance (QA) and Configuration Management personnel. The system prioritizes real-time feedback, automated investigation, and root cause identification, effectively minimizing manual effort and significantly boosting overall software quality.

Framework and Component Architecture

At the core of the methodology is a modular architecture that fuses the orchestrating power of the MCP with the advanced AI capabilities of CrewAI. The key components include:

MCP (Master Control Panel):

- Serves as a centralized user interface for managing tasks, interacting with agents, and overseeing system operations.
- Acts as middleware to facilitate robust data and command flow between users, AI agents, and connected tools such as the RAG (Retrieval-Augmented Generation) database.
- Handles event management, notifications, and user-triggered alerts.

CrewAI Data Integration Layer:

- Integrates data from multiple sources, including GIT, JIRA, CAMEO, and the RAG database, through CrewAI's connectors, ensuring real-time information flow.(JIRA, CAMEO CI/CD not wired in MVP)
- Processes incoming defect data and user inputs for in-depth analytics and decision support.

Agent Management Layer:

- Enables creation and modification of intelligent agents through the MCP, enhancing adaptability and efficiency.
- CrewAI powers these agents to execute advanced cognitive tasks and resolve complex defect scenarios.

- Intelligent AI Agents:
  - Engineer Code Review Agent: Analyzes code for defects and provides context-specific suggestions.
  - Root Cause Investigator Agent: Uses natural language processing to interpret defect reports and recommend improvement strategies.
  - Defect Discovery Agent: Detects patterns and predicts systemic issues through advanced analytics.
  - Requirements Tracing Agent: Ensures verification and traceability of requirements against test outcomes.
  - Performance Metrics Agent: Visualizes performance metrics and monitors system health.
  - Audit Agent: Conducts compliance checks and ensures adherence to standards.

  Task Automation Module:(Feature is not fully implemented in MVP proof of concept)
  - Automates task scheduling and management based on user-defined triggers (immediate, daily, or weekly) through MCP and CrewAI automation features.
  - Links tasks to RAG data for efficient tracking and follow-up.
  - Continuous Feedback and Learning Mechanism:
  - User feedback is collected via the MCP and used to continuously refine agent behavior and recommendations using CrewAI's learning algorithms.
  - Agents evolve dynamically, adapting to new patterns and suggestions over time.

Key Differentiators

1. Autonomous Multi-Agent Orchestration vs. Single-Point Assistance

- **Traditional AI Tools (Copilot, etc.)**: Provide *reactive* assistance at a single point in the development workflow—suggesting code completions, answering questions, or fixing vulnerabilities when prompted.
- **This Framework**: Implements a **Master Control Panel (MCP)** that orchestrates multiple specialized agents working *autonomously* on complex, multi-step tasks. Agents collaborate, delegate work, and coordinate through a task queue with distributed locking, enabling parallel execution across processes and containers.

2. RAG-Powered Contextual Intelligence vs. Generic Code Patterns

- **Traditional Tools**: Rely on pre-trained models and limited context windows; suggestions are based on statistical patterns from training data.
- **This Framework**: Uses **Retrieval-Augmented Generation (RAG)** with Qdrant vector store to:
    - Ingest and index the *specific* codebase, documentation, JIRA issues, and requirements (40+ repos in the roadmap)
    - Provide agents with **project-specific context** retrieved from vector embeddings
    - Enable agents to answer questions and make decisions based on the actual architecture, not generic patterns

3. Task Lifecycle Management vs. Ephemeral Interactions

- **Traditional Tools**: Interactions are stateless—each suggestion or chat is independent.
- **This Framework**: Implements full **task lifecycle tracking**:
    - Persistent Task and Activity models in MySQL/SQLite
    - Real-time status updates via Server-Sent Events (SSE)
    - Audit trails showing which agents worked on what, when, and what they produced
    - Duplicate-run protection with Redis-based distributed locks

4. Role-Based Access Control (RBAC) vs. Universal Access

- **Traditional Tools**: Same capabilities for all users.
- **This Framework**: Implements **fine-grained RBAC**:
    - Admin, Staff, and User roles with different permissions
    - OAuth SSO integration (GitHub/GitLab) for enterprise identity
    - Bearer token-based API auth with role-specific endpoints
    - Planned: staff can only delete their own agents; admins can delete any

5. Scheduled & Background Execution vs. On-Demand Only (not fully implemented in MVP proof of concept)

- **Traditional Tools**: Work only when invoked in the IDE or chat interface.
- **This Framework**:
    - **Celery + Redis** task queue for durable background jobs
    - Scheduled ingestion (cron-like daily/weekly) for keeping vectors fresh
    - Webhook-triggered ingestion (GitHub push events)
    - Asynchronous agent runs that continue even if the user disconnects

6. Observable & Instrumented vs. Black Box

- **Traditional Tools**: Limited visibility into decision-making or processing.
- **This Framework**:
    - **Prometheus metrics** (/metrics) for monitoring agent runs, queue depth, lock conflicts
    - Structured activity logs persisted to DB
    - SSE streams for real-time progress visibility
    - Sentinel files (/tmp/run_agents_*.log) for testing and debugging lock behavior

7. Customizable Agent Ecosystem vs. Fixed Capabilities

- **Traditional Tools**: Users get what the vendor provides.
- **This Framework**:
    - Define custom agents with specific tools and capabilities
    - Agent selection logic (heuristics + lightweight classifier) chooses the best agent for each task
    - CrewAI adapter allows plugging in crew-based multi-agent workflows
    - Extensible: add new agents without modifying core MCP

8. Self-Hosted & Data-Sovereign vs. Cloud-Dependent

- **Traditional Tools**: Proprietary cloud services; the code/data leaves the environment.
- **This Framework**:
    - Fully self-hosted via Docker Compose
    - Optional OpenAI mock for deterministic testing without external API calls
    - The codebase and vectors stay in the infrastructure (Qdrant runs locally)
    - Configurable: use real OpenAI embeddings or deterministic fallbacks

9. Cross-Process Coordination vs. Single-Session Context

- **Traditional Tools**: Limited to the current IDE session or chat thread.
- **This Framework**:
    - Distributed lock acquisition across FastAPI workers and Celery processes
    - Redis pub/sub for cross-container event broadcasting
    - Worker processes can continue tasks even if the web server restarts
    - Designed for horizontal scaling with shared state in Redis + DB

10. Testing & CI-First Design vs. Production-Only Focus

- **Traditional Tools**: Testing coverage varies; mostly designed for production use.
- **This Framework**:
    - Comprehensive test suite: unit tests, lock smoke tests, E2E integration scripts
    - Deterministic mocking (OpenAI, Qdrant) for reliable CI
    - GitHub Actions workflows with container-based testing
    - Playwright UI test scaffold for validating user-facing flows
    -

Table 2: Key Differentiator Summary Table

Dimension	AI-Assisted Tools (Copilot, etc.)	This Agentic Framework
Interaction Model	Reactive, single-point assistance	Autonomous multi-agent orchestration

Dimension	AI-Assisted Tools (Copilot, etc.)	This Agentic Framework
Context Source	Pre-trained patterns + local file context	RAG: project-specific vectors (repos, docs, issues)
State Management	Stateless, ephemeral	Persistent tasks, activities, audit trails
Access Control	Universal access	RBAC with OAuth SSO, role-specific permissions
Execution Mode	On-demand only	Scheduled, background, webhook-triggered
Observability	Limited	Prometheus metrics, SSE streams, activity logs
Customization	Fixed capabilities	Extensible agent ecosystem, custom tools
Deployment	Cloud SaaS	Self-hosted, data-sovereign
Coordination	Single session/IDE	Cross-process, distributed locks, pub/sub
Testing	Variable	CI-first with mocks, smoke tests, E2E scaffolds

Bottom Line

Traditional AI-assisted coding tools are **point solutions** that provide intelligent suggestions *within* the IDE workflow. This framework is a **platform** for building and orchestrating autonomous agents that can:

- Execute complex, multi-step tasks end-to-end
- Learn from a specific codebase and documentation
- Coordinate across multiple agents, processes, and containers
- Run scheduled jobs, respond to webhooks, and operate in the background
- Provide enterprise-grade RBAC, observability, and auditability
- Remain fully under the control in the organization infrastructure

It shifts from **"AI suggests code"** to **"AI agents autonomously complete tasks using the organization's knowledge." See appendix for Use Case Analysis**

Development Phases and Work Schedule

The implementation methodology is structured in distinct, iterative phases for a full enterprise project to ensure effective integration and continuous improvement:

Phase 0: Concept Proof

- Data extraction and vectorization of a subset of GitHub and requirements data into a RAG database.
- Static homepage setup for agent/task staging and prompt submission.
- LLM (Large Language Model) setup, optimization, and evaluation for prompt handling and recommendation generation.

Phase 1: Research and Requirements Gathering (3 Weeks)
- Conduct stakeholder interviews focused on MCP and CrewAI roles.
- Analyze current defect management systems to identify enhancement opportunities.
- Develop detailed design documents and obtain stakeholder approval.

Phase 2: Rapid Prototype Development (6 Weeks)
- Install MCP as the main interface and middleware, integrating CrewAI.
- Develop basic CrewAI-powered agents and perform initial system testing.

Phase 3: Advanced Agent Functionality Development (8 Weeks)
- Enhance agent functionality with predictive analytics and NLP via CrewAI.(Appendix E)
- Deploy specialized agents and finalize task automation modules.
- Conduct integration testing for MCP, CrewAI, agents, and external tools.

Phase 4: User Testing and Feedback Incorporation (5 Weeks)
- Gather user feedback on MCP and CrewAI agent interactions.
- Refine the interface and agent functionalities based on testing outcomes.

Phase 5: Deployment and Training (3 Weeks)
- System deployment with full MCP and CrewAI feature set enabled.
- User training sessions to maximize system utility and adoption.

Expected Benefits

This comprehensive methodology ensures that the system not only automates and optimizes defect detection and resolution but also provides scalable benefits throughout the software lifecycle. Expected outcomes include:

- Accelerated software quality assurance processes through real-time, intelligent defect analysis and resolution.
- Greater integration with existing development tools, minimizing disruption to current workflows.
- Continuous system improvement driven by user feedback and CrewAI's adaptive learning capabilities.
- Enhanced traceability, compliance monitoring, and actionable insights for both developers and QA teams.
- A seamless user experience with the MCP as a single touchpoint for interaction, task management, and feedback collection.

By leveraging an orchestrated combination of MCP and CrewAI, the methodology positions the Intelligent Defect Analysis and Resolution System as a next-generation solution for advancing software development and quality assurance efficiency.

Chapter 4: System Implementation

4.1 System Architecture

The Agent Task Manager system is architected using a modular, component-based approach, leveraging modern web development frameworks to facilitate scalability, maintainability, and seamless integration with AI model training pipelines. This section outlines both the conceptual and technical architecture, describing how each part interconnects to support agent and task management functionalities, while ensuring extensibility for future AI enhancements.

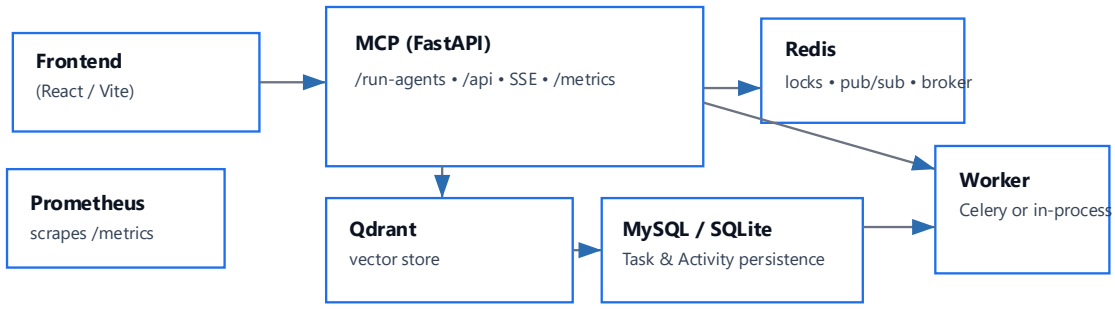
Project Structure Overview

MVP Design Summary — Intelligent Orchestration Framework Overview

Purpose: Minimal intelligent orchestration framework that accepts task requests, runs agent workflows, persists task and activity state, and prevents duplicate concurrent executions across API and worker processes.

Primary goals: reliable cross-process duplicate protection, deterministic observability for concurrency behavior, and an extensible agent execution pipeline supporting both in-process and durable worker backends.

Figure 1: Deployment Diagram



Deployment diagram — Frontend → MCP → Redis / DB / Qdrant. Workers coordinate via Redis/Celery. Prometheus scrapes MCP.

Container Requirements

The framework relies on several containers. For agent runs, SQL is required as the authoritative task/activity store.

- Required for agent operation:
  - `mysql`: SQL database used by agents to persist and retrieve tasks and activities (required; set via `DATABASE_URL`)
  - `mcp`: FastAPI backend and orchestration API
  - `worker`: Celery worker executing background jobs and agent tasks
  - `redis`: Message broker and locks for duplicate/run control
  - `qdrant`: Vector database for semantic retrieval

Infrastructure & Deployment

- Containers & services: mcp (FastAPI backend), optional worker (Celery or in-memory), redis (locks + optional pub/sub + Celery broker), qdrant (vector store), and SQL persistence (MySQL/SQLite). Prometheus scrapes /metrics.

- Build model: mcp is built into an image during development; code changes require rebuilding the image. Use bind mounts for faster local iteration if desired.

- Env
  configuration: REDIS_URL, CELERY_BROKER_URL, TASK_LOCK_TTL, TEST_HOLD_SECONDS, QDRANT_URL, OPENAI_API_KEY, and RBAC tokens.

Core Components

- API / Orchestrator (mcp): FastAPI app exposing /run-agents, /api/tasks, /events/tasks/{task_id} (SSE), /metrics, and RAG endpoints (similarity search, config). Persists through SQLAlchemy models (Task, Activity, Agent).

- MasterControlPanel: orchestrates agents, executes pipeline, publishes events via a publisher callback for SSE and persistence.

- Task Queue: pluggable; Celery-backed for durable processing (when CELERY_BROKER_URL present) or an in-memory TaskQueue for local runs.

- Redis Lock Helper: mcp/redis_lock.py implements tokenized locks for both async (redis.asyncio) and sync (redis-py) clients, including safe release logic.

- Vector store & embeddings: Qdrant for vectors; embeddings via OpenAI when available with a deterministic fallback for repeatable tests.

Intelligence Extensions

Agent Grounding & Determinism

- Layered system prompt: base operational constraints + injected contextual summaries (recent changes, test artifacts, failure clusters).

- Deterministic temperature: OPENAI_DEFAULT_TEMPERATURE=0.0 ensures reproducible reasoning in CI/regression scenarios.

- Structured output expectations: agents guided to produce machine-parseable sections (JSON blocks or delimited summaries) to reduce parsing ambiguity.

- Context hygiene: cap total tokens for artifact summaries; priority ordering (recent failures > high-severity coverage gaps > historical diffs).

- Traceability: composed grounding prompt (with secrets redacted) logged per run for auditability.

Test Artifact Intelligence

- Sources: JUnit XML (results), coverage reports (line/function gaps), execution logs (error traces), optional performance summaries.

- Normalization pipeline: parse → extract tuples (test name, status, duration, failure snippet) → cluster similar failures → summarize → embed & store.

- Attachment flow: on agent invocation, relevant artifact summaries (by file path, failing tag, coverage deficit) prepended to agent context.

- Benefits: accelerates failure triage; enables proactive suggestions (highlight flaky tests, untested critical modules).

- Metrics (extension): ingestion counters increment per batch; future cluster metrics planned.

Incremental Repository Synchronization

- Triggers: post-commit hook, polling scheduler, or manual sync endpoint.

- Flow:

1. Gather git diff (added/modified/deleted paths).
2. Chunk changed files (size threshold + overlap strategy).
3. Embed new/modified chunks → Qdrant upsert keyed by file_path:chunk_index:commit_hash.
4. Handle deletions via tombstone or soft delete (deleted=true).

- Advantages: reduced embedding cost vs full reindex; preserves semantic evolution history.
- Consistency safeguards: commit metadata table; periodic audit comparing HEAD state to stored vectors.

Concurrency & Duplicate Protection

- Multi-layer protection:
    1. Up-front Redis existence check (task:{id}:lock) — fast conflict rejection (async get or sync get via executor).
    2. Redis tokenized lock acquisition — async preferred, sync fallback in thread executor. Locks carry a token and TTL to allow safe release.
    3. DB-level atomic UPDATE fallback — set status='running' only when the row is not already running; if update touches 0 rows, respond 409.
- Workers use the same lock semantics to ensure cross-process exclusivity.
- Release occurs in finally blocks; TTLs and DB fallback mitigate leaks. Consider lease renewal for long-running runs.

Observability & Testing Instrumentation

- Structured log markers: RUN_AGENTS_DBG entries at key decision points (payload receipt, up-front check, lock attempts, acquisition/release).

- Deterministic sentinels (file-based) for smoke tests inside container:
    - /tmp/run_agents_entered.log — handler entry (timestamp + id)
    - /tmp/run_agents_lock_acquired.log — lock acquired records (method, token length)
    - /tmp/run_agents_lock_conflict.log — lock conflict records (up-front or attempted acquisition) These are simple, robust signals for automated tests and CI (less affected by log buffering).
- Metrics: TASKS_ENQUEUED, AGENT_RUNS, INGEST_COUNTER exposed at /metrics for Prometheus.
- Extended metrics (planned/adding): ARTIFACT_INGEST_BATCHES (artifact batches processed), INCREMENTAL_SYNC_COMMITS (commit-driven sync executions), VECTOR_CHUNKS_ACTIVE (active non-tombstoned chunks gauge).

Failure Modes & Hardening

- Race window: up-front check is helpful but not sufficient; tokenized lock + DB atomic update close races.
- Redis unavailability: fallback to DB atomic update; lock helper tolerates missing async client by using a sync client in an executor.
- Lock leaks: mitigated via TTL; for long-running tasks add heartbeat/lease renewal.
- Developer iteration: because the code is copied into mcp image at build, frequent rebuilds slow feedback; recommend bind mounts for active dev.

Runtime Reliability & Dev Tooling

Dev Server Persistence

- Supervisory script: scripts/start_vite_persistent.ps1 manages Vite (npm run dev) with configurable restart cap (MaxRestarts=200).

- Behavior: logs each restart; exits gracefully after cap to prevent runaway loops.

- Security context: elevation + antivirus exceptions (e.g., BitDefender) stabilize process longevity.

Port Stability & Fallback

- Standard dev port: 5173; previous fallback 5174 when conflict detected.

- Stability achieved post security exception + elevation; monitoring via scripts/check_services.ps1.

Production Asset Delivery

- Multi-stage Docker builds and serves compiled dist/ via NGINX (port 3000), fixing earlier missing asset blank screen.

Scheduled Automation

- Optional Scheduled Task registers persistent script at logon (RunLevel=Highest) applying restart cap for unattended continuity.

Developer UX & CI

- Smoke tests: concurrent POSTs with TEST_HOLD_SECONDS can force deterministic conflict; tests read sentinel files to verify one LOCK_ACQUIRED and one LOCK_CONFLICT outcome.

- Local runs: in-memory TaskQueue enables running end-to-end without Celery.

- Determinism: deterministic embedding fallback allows reproducible similarity tests without external API keys.

Security & Operations

- See Appendix D

- RBAC: sample agents/rbac.json for local dev; production should use secure secrets and proper OAuth/JWT flows.

- Secrets: keep keys in .env or Docker secrets / vault in production.

- Lock safety: tokenized locks ensure only token-holder releases; TTL prevents permanent lock hold by crashed processes.

Extensibility & Next-Phase Features

- Lease/heartbeat for long-running tasks.

- Centralized Redis pub/sub for cross-instance SSE and coordination.

- Structured JSON logging and correlation IDs (task_id) for traceability.

- CI: add a job that builds mcp, runs sentinel-based smoke tests in Docker Compose, and asserts sentinel outputs.

- Artifact change impact analysis: correlate new failures with recent code embeddings to prioritize remediation.

- Adaptive agent personas: dynamic prompt shaping based on artifact clusters (e.g., "test stabilizer", "coverage extender").

- Proactive anomaly detection: compare failure cluster embeddings vs historical baselines for early regression flags.

- Commit-to-context trace: surface commit metadata and diff summary within agent grounding prompt.

- Automated documentation refresh: trigger PDF conversion on designated docs after structural changes (CI job integration).

Minimal Recommended Next Steps (easiest first)

1. Stabilize and run sentinel smoke test with TEST_HOLD_SECONDS=30 and assert sentinels show one acquired + one conflict.

2. Add lease renewal/heartbeat for long runs to avoid TTL races.

3. Move logs to structured JSON and include a task_id correlation field.

4. Add CI job to build mcp and run smoke tests; assert sentinel files.

5. Harden production deployment: use Docker secrets, add health checks and alerting on Redis or repeated 409s.

6. Change abstraction of MCP to actual **Model Context Protocol** implementation

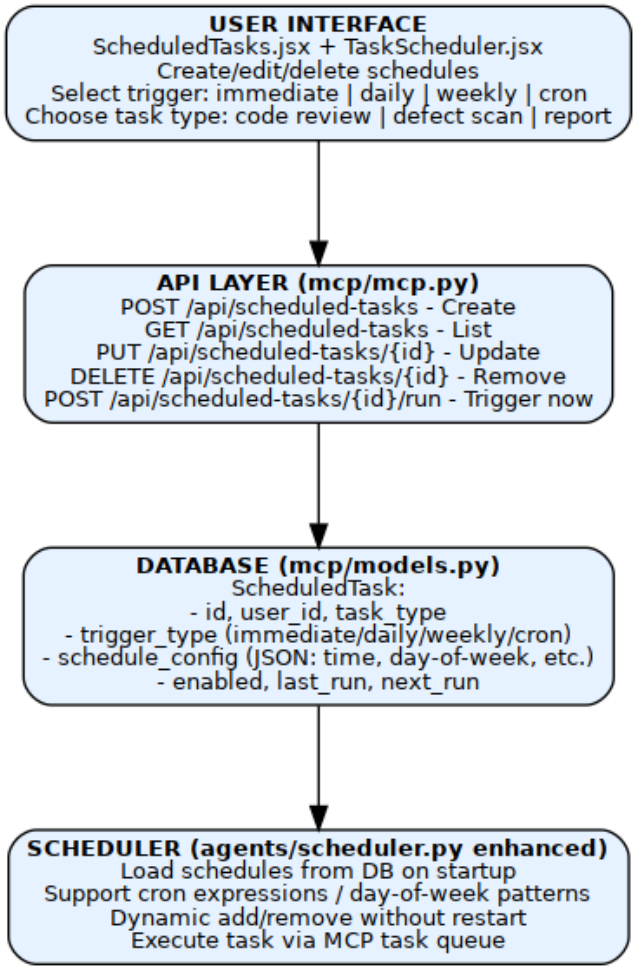
7. Fully implement automation, scheduling and UI interfaces

8. JIRA: OAuth/API client, issue sync, webhook-based artifact ingestion.

9. CAMEO: API/format adapters, requirement-to-code trace ingestion, change impact reports.

10. CI/CD: Stable pipelines for smoke/E2E/UI, coverage gates, artifact publishing, and Prometheus metrics export for agent durations/delegation phases.

Figure 2 Enterprise Model Diagram



System Integration and Extensibility

The architecture anticipates integration with existing development tools and external services. The modular React design ensures that components are reusable and easy to connect to RESTful APIs, databases, and AI algorithms. Privacy and security are considered at every layer, particularly with input validation and the access control structure embedded in the task creation workflow. As the project evolves, new components—for example, feedback modules, model evaluation dashboards, and data visualization panels—can be seamlessly added to the components/ directory and integrated into App.js.

Summary

This robust system architecture, built around clear separation of concerns and enhanced component schematics, serves as the foundation for an adaptive, AI-driven agent task management platform. It is primed for future enhancements, enabling the integration of

sophisticated AI models, prioritization algorithms, and secure data handling practices to support defect resolution and continuous improvement.

The system is implemented using a modular architecture to facilitate integration with existing development tools. Key components include data ingestion pipelines, the AI model training environment, and a user interface for interacting with the system. Special attention is given to data privacy and security throughout the implementation phase. User feedback mechanisms are incorporated to enable continuous learning with a planned adaptive feedback loop and system refinement.(Appendix E) The deployment process ensures that the system operates efficiently in a real-world software development environment.

Chapter 5: Results and Evaluation

This chapter maps the currently implemented end-to-end (E2E) and smoke tests to core system components, highlights coverage gaps, and proposes targeted next tests. It focuses on practical scenario coverage rather than line/branch metrics.

- E2E and smoke coverage is strong for: agent runs, ingestion (incl. incremental), OpenAI adapter, and Qdrant integration.
- Primary gaps: health/metrics endpoints, Similarity Search API, RAG config API, admin ingest API at the endpoint layer, webhooks, SSE events, and focused DB unit tests.
- The proposed next tests close operational and API-surface gaps with minimal scaffolding.

Highlights:

Evidence Summary of Multi-Agent parallel execution

- Parallel Execution: agents.py uses asyncio.create_task for each agent and await asyncio.gather(*agent_tasks), enabling concurrent processing.
- Instrumentation : Per-agent start/end/duration plus coordination metrics (wall_time, sum_agent_durations, parallel_efficiency_ratio, overlap_time) captured under results["__coordination_metrics__"].
- Demo Run Result: Wall time ≈ 3.09s vs sum of agent durations ≈ 3.11s (ratio ≈ 1.00), indicating agents overlapped almost fully (EngineerAgent did the long work; others finished quickly in the same window).
- Overlap Metric: overlap_time ≈ 0.015s (difference between summed durations and wall time) confirms concurrent scheduling rather than sequential execution.
- Agent Timings (sample): EngineerAgent ~3.09s, other agents ~2–4ms each, all starting within the EngineerAgent window.

Coordination Metrics Keys

- wall_time: Total elapsed for all agents.
- sum_agent_durations: Sum of individual agent durations (would be higher than wall time if parallel).

- parallel_efficiency_ratio: sum_agent_durations / wall_time (≈1 implies high concurrency; >>1 indicates strong parallel speedup).
- overlap_time: Extra time saved vs sequential (sum_agent_durations - wall_time).
- agents: Per-agent timing dict (start, end, duration).
- agent_count: Total number of agents participating.

Evidence Summary of Agent Delegation Demo Results

- Phase 1: Engineer review suggests files auth.py, pipeline.py.
- Phase 2: Parallel delegation
  - RootCause: “Likely cause identified in module X.”
  - DefectDiscovery: “Found potential systemic issues in the CI pipeline.”
- Phase 3: Parallel finalization
  - Requirements: “Mismatch found between requirement Y and implementation.”
  - Metrics: “All performance metrics are within the acceptable range.”
  - Audit: “Compliance check passed.”
- Provenance: Delegation recorded
  from EngineerAgent → RootCauseAgent, DiscoveryAgent with context
  keys files, engineer_notes

Covered: agent run flow (/run-agents), lock duplicate protection, ingestion (incl. incremental sync + Qdrant deletes), OpenAI adapter, Qdrant integration. Optional UI smoke included. Gaps: /health and /metrics, /similarity-search, RAG config (POST/GET), /admin/ingest at the endpoint layer (pipeline covered), /webhook/github and /webhook/jira, SSE events endpoint, DB unit tests for IndexedCommit. Next tests: compact skeletons provided for each missing area with suggested fixtures. Quick runs:
Unit/integration: pytest -q E2E mock: python run_e2e_integration.py --mock Smokes: python scripts/run_agent_smoke2.py; python run_crewai_smoke.py Lock smoke (PowerShell): run_lock_smoke.ps1 -TaskId 11

Component Test Coverage Map

Component	Key Endpoints / Functions	Covered By	Status
Agent run orchestration	POST /run-agents, SSE events	E2E test, lock tests	Covered (happy-path + duplicate protection)

Component	Key Endpoints / Functions	Covered By	Status
Health & Metrics	GET /health, GET /metrics	—	Gap
Similarity Search API	POST /similarity-search	—	Gap
RAG Config API	POST /rag-config, GET /rag-config	—	Gap
Admin Ingest	POST /admin/ingest	Incremental sync covered at script level; endpoint not directly	Partial (gap at API layer)
GitHub Webhook	POST /webhook/github	—	Gap
JIRA Webhook	POST /webhook/jira	—	Gap
Task Events (SSE)	GET /events/tasks/{task_id}	E2E (implicitly via run)	Partial
RAG Admin View	GET /rag-admin	—	Gap
Ingestion Pipeline	scripts/ingest_repo.py	Ingest unit tests + incremental sync E2E	Covered
Qdrant Integration	Upsert/delete, filters by metadata	Incremental sync test, CI smoke	Covered (key paths)
DB Models	IndexedCommit	Incremental sync test (persistence)	Partial (no unit tests)

Component	Key Endpoints / Functions	Covered By	Status
OpenAI Adapter	agents/crewai_adapter.py	Unit test + smokes	Covered
Agents (CrewAI)	Orchestration, prompts	Unit test + smokes	Covered (basics)
Frontend UI	Query → response path	Playwright smoke (optional)	Partial
RBAC	rbac.json;	light via endpoint-level calls and E2E scaffolds; no dedicated unit-test suite reported	Gap
Distributed Locks	sentinel smoke test (run_lock_smoke.ps1), duplicate-run validation, metrics observation; reliable for concurrency paths.	sentinel smoke test (run_lock_smoke.ps1	Covered
CI Integration	run_lock_smoke.ps1; run_e2e_integration.py (mock/real); test_ui_agent_flow.py; mock server openai_mock.py / run_openai_mock.py	smoke/E2E	Covered

Chapter 6: Conclusion and Future Work

The multi-source RAG architecture (Git + Qdrant + LLM) is a SIGNIFICANT ADVANTAGE for intelligent agent frameworks in code analysis contexts.
The synergy between:
-**Git's temporal/change data**
-**Qdrant's semantic search**
-**LLM's reasoning capabilities**

creates an emergent intelligence that exceeds the sum of its parts. Each source fills gaps the others cannot, resulting in a system capable of answering a dramatically wider range of questions with higher accuracy and richer context.
Multi-Source RAG an Advantage for Intelligent Frameworks

1.**Completeness**: Answers questions single-source RAG cannot
2.**Accuracy** Verifiable facts (git) + semantic understanding (vector DB) + reasoning (LLM)

3.**Adaptability**: Different queries use optimal data sources
4.**Evolution Analysis**: Understands code *over time*, not just current state
5.**Trust**: Cryptographically verified history + similarity search
6.**Context Richness**: LLM receives multi-dimensional context

When Multi-Source Shines:

✅ **Code review automation**: "What changed and why?"
✅ **Incident analysis**: "When was this bug introduced?"
✅ **Refactoring assistance**: "Find similar patterns that need updating"
✅ **Onboarding**: "Show me recent changes to auth module"
✅ **Compliance**: "Who approved this security change?"
✅ **Impact assessment**: "What else changed in related commits?"

This capstone project highlights the potential of AI-driven solutions in enhancing software defect management. The implemented system has potential for significant improvements in defect detection accuracy and resolution efficiency. Future work will focus on expanding the system's capabilities to handle a broader range of software artifacts and integrating advanced AI techniques for even greater adaptability. Ongoing research and user engagement will be essential to ensure the system remains effective and relevant in dynamic development environments. The next phase of evaluation of this system should focus on system performance, accuracy, or user outcomes. There should also be an effort to collect data on defect detection rates, false positives/negatives, or time-to-resolution improvements.

Appendix A POSSIBLE OR EXPECTED USE CASE ANALYSIS

Hypothetical Use Case Classification
(Not actually performed or exercised in MVP)

Statistics

IDE Assistants Can Handle:

- **Fully**: 2 use cases (7.4%)
  - #17: Doxygen documentation generation
  - #22: Basic test case generation
- **Partially**: 5 use cases (18.5%)
  - #4: Current file dependency checking
  - #8: Generic troubleshooting checklists
  - #18: Single-file defect detection
  - #19: Visible code review
  - #26: Current document review

Framework Required:

- **Essential**: 20 use cases (74.1%)
  - All cases requiring Git history analysis
  - All cases requiring external system integration (JIRA, GitLab, CAMEO)
  - All cases requiring cross-file/cross-repository analysis
  - All cases requiring temporal pattern detection
  - All cases requiring team coordination

Detail

#	Use Case	Capability Required	Can IDE Do It?	Requires Framework?	Rationale
1	Insertion of expanded descriptions and history in defect reports	Git history + defect system integration	❌ No	✅ Yes	Requires access to full Git history, commit messages, and defect tracking system context—beyond single-file scope
2	Agent sends comments to team members of potential issues	Team coordination + notification system	❌ No	✅ Yes	Requires integration with communication systems (JIRA, GitLab) and team assignment logic

#	Use Case	Capability Required	Can IDE Do It?	Requires Framework?	Rationale
3	Agent suggests/assigns team members to defect reports of similar work	Historical work patterns + team database	❌ No	✅ Yes	Needs semantic search across past defects, Git blame data, and team member expertise mapping
4	Identify inconsistencies of build-tags and code dependencies	Cross-file dependency analysis	⚠️ Partial	✅ Yes	IDE can check current dependencies, but framework provides comprehensive multi-file + build config analysis
5	Checks alignment of pipeline tests and pull requests	CI/CD integration + test coverage	❌ No	✅ Yes	Requires pipeline log analysis, test mapping to code changes, and PR metadata
6	Assists planning stories/tasks and tracking completion against schedule	Project management integration	❌ No	✅ Yes	Needs JIRA/project tool integration, velocity tracking, and historical completion data
7	Suggests root cause understanding and predicts upcoming execution misses	Temporal pattern analysis + predictive modeling	❌ No	✅ Yes	Requires analyzing commit patterns over time, failure trends, and cross-repository correlations
8	Generate problem solution troubleshooting checklist	Historical defect patterns	⚠️ Partial	✅ Yes	IDE can suggest generic checklists; framework provides context-specific lists based on similar past issues

#	Use Case	Capability Required	Can IDE Do It?	Requires Framework?	Rationale
9	Identifies weaknesses in timing and sequencing of code development	Git timeline analysis + dependency ordering	❌ No	✅ Yes	Requires commit timeline analysis, branch merge patterns, and dependency graph evolution
10	Reviews logs from running code, pipeline test failures, and suggests root cause	Log analysis + historical failure patterns	❌ No	✅ Yes	Needs access to CI/CD logs, test results, and semantic search for similar past failures
11	Records suggested root causes in GitLab/database and notifies developer	External system integration + persistence	❌ No	✅ Yes	Requires write access to GitLab/JIRA and database tracking of root causes
12	Tracks requirements against identified tests and reports status	Requirements management + traceability	❌ No	✅ Yes	Needs CAMEO integration, test-to-requirement mapping, and coverage reporting
13	Checks alignment provided by Cameo to track achieved requirements	CAMEO integration + verification	❌ No	✅ Yes	Requires direct integration with CAMEO requirements management system
14	Performs or reviews system analyses and activities	Multi-source analysis + compliance tracking	❌ No	✅ Yes	Needs access to multiple data sources (Git, JIRA, logs) for comprehensive system analysis
15	Assists resolution of CI/CD blocks	Pipeline integration +	❌ No	✅ Yes	Requires CI/CD log access, failure

#	Use Case	Capability Required	Can IDE Do It?	Requires Framework?	Rationale
		historical resolution patterns			pattern analysis, and semantic search for similar resolutions
16	Generate continuous integration records for verification	CI/CD data aggregation + reporting	❌ No	✅ Yes	Needs pipeline data collection, test result aggregation, and compliance reporting
17	Doxygen supplement for compliance with coding standards	Code documentation + standards checking	✅ Yes	⚠️ Optional	IDE can generate documentation; framework adds standards compliance verification across codebase
18	Performs review for code defects across files	Cross-file static analysis + pattern detection	⚠️ Partial	✅ Yes	IDE checks current file; framework performs semantic search for similar defect patterns across all files
19	Performs code review of pull requests and commits	PR context + multi-file diff analysis	⚠️ Partial	✅ Yes	IDE reviews visible changes; framework analyzes impact across codebase + historical similar changes
20	Search RAG for previous defect behavior	Vector search + defect database	❌ No	✅ Yes	Core framework capability: semantic search across historical defects and resolutions

#	Use Case	Capability Required	Can IDE Do It?	Requires Framework?	Rationale
21	Cross-reference RAG for code reuse	Semantic code search + similarity detection	❌ No	✅ Yes	Requires vector search across entire codebase to find semantically similar implementations
22	Build test cases and border tests	Test generation + edge case analysis	✅ Yes	⚠️ Optional	IDE can generate basic tests; framework adds historical defect patterns and edge cases from past failures
23	Assess defect report for potential duplicates, common root, systemic patterns	Defect clustering + pattern analysis	❌ No	✅ Yes	Requires semantic search across all defects, temporal pattern detection, and root cause correlation
24	Prescreen defect reports and code before reviews and staff meetings	Multi-source pre-analysis + summarization	❌ No	✅ Yes	Needs access to defect reports, code changes, Git history, and test results for comprehensive pre-screening
25	Generation or second validation of Configuration status report	Configuration management + validation	❌ No	✅ Yes	Requires access to configuration database, version tracking, and compliance verification
26	Review plans and documents for inconsistency	Document analysis + cross-referencing	⚠️ Partial	✅ Yes	IDE can check current document; framework cross-references multiple documents and

#	Use Case	Capability Required	Can IDE Do It?	Requires Framework?	Rationale
					code for consistency
27	Suggests and generates new defect reports for new work	Proactive defect detection + ticket generation	❌ No	✅ Yes	Requires analyzing code changes against historical patterns and creating tickets in external systems

Appendix B **Key Differentiators Why IDE Assistants Fall Short:**

1. **No Historical Context**
    - Cannot analyze "When was this bug introduced?"
    - Cannot track "How did this code evolve over time?"
    - Cannot answer "Who changed the authentication logic?"
2. **Limited Scope**
    - Work within visible files only
    - Cannot perform cross-codebase semantic search
    - Cannot discover similar patterns in distant files
3. **No External Integration**
    - Cannot access JIRA, GitLab, or CAMEO
    - Cannot write defect reports or assign team members
    - Cannot pull CI/CD logs or test results
4. **No Team Intelligence**
    - Cannot identify expert developers for specific areas
    - Cannot track team velocity or predict completion
    - Cannot coordinate across team members
5. **Surface-Level Analysis**
    - Suggest code based on current patterns
    - Cannot explain *why* code changed historically
    - Cannot perform root cause analysis across commits

Framework Advantages:

1. **Temporal Intelligence** (Git Integration)
    - Cryptographically verified commit history
    - Author attribution and timestamps
    - Exact diffs showing what changed when
2. **Semantic Intelligence** (Qdrant Vector DB)
    - Cross-codebase semantic search
    - Pattern similarity across entire repository
    - Natural language queries for conceptual searches
3. **Multi-Source RAG**
    - Git metadata (temporal context)
    - Qdrant search results (semantic context)
    - External systems (JIRA, CAMEO, CI/CD)
    - Current code state (static context)
4. **Enterprise Integration**
    - Defect tracking systems
    - Requirements management (CAMEO)
    - CI/CD pipelines
    - Team coordination tools

Recommendations

Use IDE Assistants For:

- Code completion and generation
- Current file refactoring
- Basic documentation generation
- Simple test case scaffolding
- Syntax and style checking

Use Intelligent Framework For:

- Root cause analysis requiring Git history
- Cross-file defect pattern detection
- Requirements traceability and compliance
- Team assignment and coordination
- CI/CD failure analysis and resolution
- Historical defect search and reuse
- Configuration management validation
- Project planning and tracking
- Proactive issue detection and prevention

Hybrid Approach:

For use cases marked "Partial", use IDE assistants for immediate development tasks, but leverage the framework for:

- Comprehensive analysis before code review
- Historical context for troubleshooting
- Cross-repository impact assessment
- Quality gates and compliance verification

Appendix C **Getting Started**

Prerequisites

Before using RAG-POC, ensure users have:

- Docker and Docker Compose installed
- Access to a Git repository (local or remote)
- An OpenAI API key (recommended for production use)

Initial Setup

1. **Clone the repository:****
   ```powershell
   git clone https://github.com/reginaldrhoe/rag-poc.git
   cd rag-poc
   ```

2. **Create environment file:****
   Create a `.env` file in the root directory:
   ```env
   OPENAI_API_KEY=sk-your-api-key-here
   OPENAI_DEFAULT_TEMPERATURE=0.0
   QDRANT_URL=http://qdrant:6333
   CELERY_BROKER_URL=redis://redis:6379/0
   GIT_REPO_PATH=/repo
   ```

3. **Start the services:****
   ```powershell
   docker compose build mcp worker
   docker compose up -d
   ```

4. **Verify the setup:****
   ```powershell
   Invoke-RestMethod -Uri http://localhost:8001/health
   ```

5. **Access the web interface:****
   Open the browser to http://localhost:5173

Using the Web Interface

Overview

The web interface provides:

-**Task Dashboard**: View all tasks and their status
-**Task Creation**: Submit new analysis tasks
-Real-time Updates: Live agent activity via Server-Sent Events (SSE)
-**Settings Panel**: Configure repositories and collections

Dev UI (Vite) vs Containerized Frontend

- Dev UI (Vite): Runs at `http://localhost:5173` and is recommended during development for hot-reload and faster iteration.
- Containerized Frontend: When using Docker Compose, a frontend may be available at `http://localhost:3000`.

Start the Vite dev server (if not using the containerized frontend):

```powershell
cd web
npm install
$env:VITE_API_URL = "http://localhost:8001"
npm run dev
```

Appendix D How Security is Implemented in the MVP

-

RBAC

- Scope: Token-based RBAC for dev/test; sample policy in rbac.json.
- Enforcement: Protected endpoints (e.g., POST /admin/ingest) require Authorization: Bearer <token> with role checks in FastAPI.
- Auditability: Access is logged via API logs; sentinel/metrics focus on run behavior, not full access trails.
- Gaps: No fine-grained resource-level rules or dynamic policy engine; unit tests for RBAC are minimal.

OAuth SSO

- Status: Not implemented in 2.3.0. No OAuth/OIDC/JWT integration, login flows, refresh-token rotation, or provider federation.
- Planned: Add OAuth/OIDC with JWT validation middleware, role claims mapping, and session management in a later phase.

Data Sovereignty

- Storage: MySQL (tasks/activities), Qdrant (vectors), Redis (locks/broker). No external data egress by default when using the mock.
- External AI Calls: Can be fully disabled by using openai_mock.py and setting app configs to the mock; otherwise, real OpenAI implies data transit to the provider governed by their policies.
- Boundary Controls: .env configuration determines endpoints; artifacts served locally at http://localhost:8001/artifacts/.
- Gaps: No field-level encryption at rest, data residency controls, or DLP scanning; backup/retention policies not formalized.

Operational Safeguards

- Secrets: Expected in .env/Docker secrets; keys not hard-coded in the repo.
- Network: Localhost by default; container networks isolated via Docker Compose.
- Privacy-by-default option: Use the local mock for LLM calls to avoid any external data transmission.

Summary

- 2.3.0 provides basic RBAC with bearer tokens and local-only data paths when using the mock; OAuth SSO and advanced data sovereignty controls are future work.

Appendix E – CrewAI Continuous Learning

Current State

- current MVP version (2.3.0) does not yet implement user feedback-driven agent reweighting."
- No implemented feedback loop: Agents run statelessly per task; outputs aren't scored by users nor persisted as preference signals.
- CrewAI integration here is superficial (wrapper-style invocation, deterministic fallbacks) without adaptive weighting, fine-tuning, or historical performance tracking.
- repository implementation (code, scripts, tests) has zero mechanisms for collecting, storing, or applying user feedback as of version(2.3.0)
- SQL database has persistent data for later analysis

Gap Analysis

- Missing Data Flow: No endpoint or UI element for "thumbs up/down", rating, rationale, or corrected answer submission.
- Missing Persistence: No tables (e.g., AgentFeedback, AgentPerformanceHistory) or vector metadata linking feedback to agent outputs.
- Missing Adaptation Logic: Agent selection order and prompt shaping are static; no decay/boost based on past performance.
- Missing Metrics: No Prometheus counters/histograms for feedback counts, acceptance rates, or drift.
- Missing Transparency: DESIGN/OPERATION manuals don't state that adaptation claims are aspirational.

Evidence (What Exists vs Claimed)

- Exists: Parallel dispatch + delegation demos, timing metrics, artifact summaries.
- Absent: Any historical comparison of answer quality, reinforcement signals, retraining triggers, dynamic prompt adjustments.

Proposed Feedback Loop Architecture

1. Capture Layer
    - UI controls: "Approve", "Reject", "Partially Useful", plus optional free-text correction.
    - API: POST /feedback with payload { task_id, agent_name, output_hash, verdict, rating (0–5), correction_text?, tags? }.
2. Storage Layer
    - Tables:
        - AgentOutput: (id, task_id, agent_name, content_hash, content, created_at)
        - AgentFeedback: (id, agent_output_id, verdict ENUM, rating INT, correction_text, user_id, created_at)
        - AgentPerformanceAggregate: (agent_name, window_start, window_end, approvals, rejections, avg_rating, adjusted_weight)
3. Processing Layer (batch or streaming):
    - Increment aggregates per feedback event.

    - Recompute adjusted_weight = base_weight * f(approval_rate, avg_rating, recency_decay).
    - Store “prompt adjustment hints” (e.g., frequent correction keywords) for future prompt augmentation.
4. Adaptation Application:
    - At task dispatch: order agents by adjusted_weight; optionally skip consistently low performers.
    - Prompt enrichment: Prepend top correction patterns for the agent domain (e.g., “Be concise about performance metrics”).
5. Metrics:
    - agent_feedback_total{agent, verdict}
    - agent_rating_average{agent}
    - agent_weight{agent}
    - agent_feedback_latency_seconds (capture reaction time from output generation to feedback receipt)

Algorithm Roadmap

- Phase 1 (Instrumentation): Introduce endpoints, tables, metrics without changing agent logic.
- Phase 2 (Weighted Scheduling): Order or probabilistically sample agents based on adjusted_weight.
- Phase 3 (Prompt Personalization): Extract n-gram correction patterns → summary → inline guidance.
- Phase 4 (Adaptive Budgeting): Reduce runtime for low-performing agents (e.g., skip full analysis steps).
- Phase 5 (Human-in-the-Loop Fine-Tuning Stub): Export curated (prompt, output, feedback) triplets for external model fine-tuning pipelines.

Minimal Initial Implementation (Incremental)

- Add AgentOutput and AgentFeedback tables.
- Add publisher hook to persist each agent’s final content.
- Endpoint POST /feedback validating agent_output_id or (task_id, agent_name, output_hash).
- Aggregation job (daily or on-demand) computing approval rate & new weights.
- Configuration: store agent_weights.json; reload into AgentManagementLayer on change.

Example Table Schemas (Simplified)

- AgentOutput(id INTEGER PK, task_id INTEGER, agent_name TEXT, content_hash TEXT, content TEXT, created_at DATETIME)
- AgentFeedback(id INTEGER PK, agent_output_id INTEGER FK, verdict TEXT CHECK in ('approve','reject','partial'), rating INTEGER, correction_text TEXT, user_id TEXT, created_at DATETIME)
- AgentPerformanceAggregate(agent_name TEXT, window_start DATETIME, window_end DATETIME, approvals INTEGER, rejections INTEGER, avg_rating REAL, adjusted_weight REAL)

Feedback → Weight Function

- approval_rate = approvals / max(approvals + rejections, 1)

- score = 0.6*approval_rate + 0.3*(avg_rating/5.0) + 0.1*recency_factor
- adjusted_weight = clamp(base_weight * (0.5 + score), 0.2, 2.0)
- recency_factor decays linearly or exponentially over N days.

Risks & Considerations

- Cold Start: New agents need default neutral weight (avoid penalizing absence of feedback).
- Feedback Quality: Low-quality or malicious feedback could distort adaptation (consider minimum quorum or admin override).
- Privacy: Store only necessary correction text; avoid sensitive data leakage in prompts.
- Consistency: Adaptation frequency too high may cause behavioral instability—batch updates recommended.

Next Actions (Later Phase)

1. Patch docs to clarify current limitations.
2. Implement AgentOutput persistence + POST /feedback.
3. Add basic aggregate calculation script.
4. Instrument metrics (agent_feedback_total, agent_rating_average).
5. Introduce weight-based agent ordering.

Role of SQL Persistence

- Tasks/Activities: Stores Task and Activity history (inputs, statuses, timestamps) as ground truth for performance and drift analysis over time.
- Agent Outputs: Persist each agent's final response with content hash and metadata to create a labeled corpus for training/evaluation.
- Feedback Signals: Record user verdicts (approve/reject/partial), ratings, and corrections to turn raw outputs into learning examples.
- Artifacts & Context: Link test results, coverage, and commit diffs to each run so learning reflects real project signals, not isolated answers.
- Windowed Aggregates: Compute rolling metrics per agent/domain (approval rate, avg rating, time-to-answer) to drive adaptive behavior.

Minimal Schema (Practical)

- AgentOutput(id, task_id, agent_name, content_hash, content, created_at)
- AgentFeedback(id, agent_output_id, verdict ENUM, rating INT, correction_text, user_id, created_at)
- AgentPerformanceAggregate(agent_name, window_start, window_end, approvals, rejections, avg_rating, adjusted_weight)
- Optional: AgentWeights(agent_name, base_weight, adjusted_weight, updated_at)

How Continuous Learning Emerges

- Capture: Persist every agent output; attach user feedback via POST /feedback (verdict, rating, correction).
- Aggregate: Nightly job computes approval rate, average rating, recency-decayed scores per agent/domain.
- Adapt: Update AgentWeights and prompt hints (common correction patterns); reorder/skip agents and personalize prompts on next runs.

- Evaluate: Track acceptance rates and latency pre/post-adaptation; roll back if metrics regress.
- Curate: Export (prompt, output, feedback) triplets as a training set for offline model fine-tuning.

Example Queries

- Recent approval rate:
-
- Update adjusted weights:
-
- Insert feedback:
-

Adaptation Application

- Weighted Scheduling: AgentManagementLayer orders agents by adjusted_weight; optionally gates low performers.
- Prompt Personalization: Prepend top correction n-grams (concise, domain-specific) to agent prompts.
- Budgeting: Reduce runtime or depth for agents with persistently low scores; increase for high performers.

Privacy & Governance

- Minimize stored PII in correction_text; redact sensitive substrings.
- Retention policies for feedback and outputs; allow opt-out per task/user.
- Use the local mock LLM to keep data sovereign; avoid external calls when required.

Next Steps to Enable

- Add AgentOutput persistence hook and POST /feedback endpoint.
- Implement nightly aggregate job and AgentWeights consumption in dispatch.
- Expose metrics:
- agent_feedback_total, agent_rating_average, agent_weight via /metrics.