LLM Powered Postmortem Automation for Site Reliability Engineering Teams

Course No : CC ZG628T

Dissertation

Student Name:
Pawar Abhay Dnyandeo

BITS ID: 2022MT03552

Submitted this as MID TERM REPORT as part of partial fulfilment of

Degree Program:
Master of Technology in Cloud Computing

Research Area:
Site Reliability Engineering, Postmortem Practices and Multi-Agent Systems

Dissertation / Project Work carried out at:
Integra Micro Software Services
IMSS Pvt Ltd, Bengaluru

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE,

PILANI - RAJASTHAN - 333031

(First Semester 2025- 2026)

Sept 15, 2025

Abstract

Modern software platforms are predominantly composed of distributed microservices, which significantly increase operational complexity during system failures. Site Reliability Engineers (SREs) must correlate extensive logs, trace code paths, diagnose root causes, recommend remediations, and compile postmortems—all under time pressure. This manual workflow contributes to elevated Mean Time To Resolution (MTTR), adversely impacting business continuity, team efficiency, and inter-team collaboration.

This report presents the design and development of a **Graph-Powered, Model Context Protocol based Platform - Multi-Agent Collaborative System created using CrewAI** aimed at reducing MTTR through automation. The framework introduces a novel **Model Context Protocol (MCP)**, which departs from conventional Retrieval-Augmented Generation (RAG). Instead of vector-based semantic retrieval, MCP constructs a **Code Intelligence Graph** using **Abstract Syntax Tree (AST) analysis** to represent structural and causal relationships within application's source code. This enables precise and explainable reasoning by crew ai based specialized AI agents.

The platform follows a decoupled client–server architecture: a Flask backend orchestrates three CrewAI agents—Diagnostician (Root Cause Analysis), SRE Engineer (Remediation Planning), and Technical Writer (Report Generation)—while a Gradio frontend provides an interactive user interface. A high-fidelity incident simulator, driven by a deliberately faulty application, generates realistic logs, metrics, and impact signals to test scenarios such as database deadlocks, slow queries, API mismatches, configuration errors, and performance bottlenecks.

The operational prototype created demonstrates strong accuracy, auditability, and reduced cognitive load for incident handling. The results highlight the feasibility of AI-assisted postmortem tools and suggest future enhancements through continuous graph updates and enriched telemetry integration.

Table of Contents

1. Introduction

1.1. The Real World Problem in Modern SRE: The Challenge of MTTR

As software architectures evolve into highly distributed microservice ecosystems, the cognitive demands on Site Reliability Engineers (SREs) during production incidents have intensified. Engineers often need to examine thousands of log entries across multiple services, correlate them with relevant sections of source code, and mentally reconstruct the potential failure chain—all while operating under severe time pressure. This investigative load directly impacts one of the most critical reliability metrics: **Mean Time to Resolution (MTTR)**. Elevated MTTR results not only in revenue loss and erosion of customer trust but also contributes to operational fatigue and burnout within engineering teams.

1.2. The Inadequacy of Existing Paradigms for Code-Level Analysis

The emergence of Large Language Models (LLMs) has opened new opportunities for us to automating diagnostic tasks. However, their practical application to code-level analysis remains limited. The prevailing method, **Retrieval-Augmented Generation (RAG)**, provides external knowledge by retrieving semantically similar text. While useful for documentation lookup, **RAG is structurally blind**: it cannot capture the syntactic and causal relationships that govern our program behavior.

For instance, when investigating a database deadlock, a conventional RAG system may retrieve text discussing concurrency or functions that handle database connections. Yet, it typically fails to uncover the true cause—two distinct functions acquiring locks on multiple resources in inconsistent order. In such cases, semantic correlation does not equate to causal explanation. Similar issues arise with configuration errors or API mismatches, where the absence or misplacement of checks at specific code points is often decisive.

The core limitation lies not in the LLM's reasoning ability but in its context retrieval: **semantic similarity ≠ causal relevance**.

1.3. Thesis: A Novel Framework for Causal, Agentic Analysis

This work argues that effective automated diagnosis requires a retrieval mechanism aligned with the **structural and causal nature of software systems**. The central thesis is that a **Code Intelligence Graph**, constructed through **Abstract Syntax Tree** (AST)-based static analysis and domain mappings, provides a superior foundation for context retrieval than flat vector indices.

By coupling this graph with a **Model Context Protocol (MCP)** and a **multi-agent collaborative workflow**, we enable crewai based AI agents to traverse program structures, reconstruct causal chains, and generate transparent, explainable reasoning. Unlike conventional RAG, this approach elevates context from “bags of semantically similar tokens” to **typed, queryable relationships**, ensuring that reasoning begins with the right evidence.

1.4. Core Contributions

- Graph-Powered Model Context Protocol (MCP) with crewai based agents as a custom local tool: A paradigm that replaces vector search with a queryable, causally structured knowledge graph.

- Domain-Specific Multi-Agent Workflow: A crewai agent pipeline of the **three specialized roles—Diagnostician** (Root Cause Analysis), **Engineer** (Remediation), and **Writer** (Report Generation)—designed to mirror the SRE workflow.

- End-to-End Simulation and Analysis Platform: A fully functional prototype encompases incident generation, graph ingestion, multi-agent reasoning, and automated postmortem reporting.

2. System Architecture: A Decoupled, Multi-Component Design

The proposed framework is designed as a **decoupled mcp client–server system** means it is powered by the **Model Context Protocol based Platform - Multi-Agent Collaborative System created using Crewai workflow as a tool**.

This architecture balances modularity, scalability, and explainability while supporting the complete postmortem lifecycle—from incident simulation to diagnosis, remediation, and report generation.

2.1. Architectural Philosophy: The Decoupled MCP Client-Server Model

A fundamental design choice was to **separate the user-facing interface and something like a chatbot where user can select which tasks he wants to be completed by agents with custom input - instead of some existing the computationally intensive AI orchestration** technique which are irrelevant to our usecase.

- **Client (Gradio Frontend – mcp_host_gradio.py)**
  Responsible solely for user interaction, the client manages log or file ingestion, user controls (e.g., analysis depth), streaming display of intermediate outputs, and artifact downloads.

- **Server (Flask Backend – mcp_server.py)**
  Orchestrates the CrewAI agents, manages tool access, streams intermediate results, and assembles the final report. Resource-heavy workloads execute here, ensuring the client remains lightweight and responsive.

A key design decision was to separate the user interface from the computationally intensive AI workload. This client-server model provides significant advantages:

- **Separation of Concerns**: The Gradio frontend client (mcp_host_gradio.py) is solely responsible for user interaction, while the Flask server (mcp_server.py) handles all the complex logic of crew ai based agent orchestration and LLM communication.
- **Scalability**: In future times when more and more tools are added to the server, it can be deployed on powerful hardware, while the client remains a lightweight web interface accessible from any browser.
- **Modularity**: The model-agnostic llm_provider.py can be updated on the server without requiring any changes to the client, as we demonstrated during

development.

2.2 The Model Context Protocol (MCP)

MCP is the **structured context-retrieval protocol** underpinning the system. Unlike traditional RAG, which relies on semantic similarity, MCP retrieves **causal and structural evidence** from a **Code Intelligence Graph**. It operates in three stages:

We selected **OpenRouter API R1T2 Chimera Free Model API key**, later we can upgrade to Gemini or Anthropic Claude Models

- **Acquisition (Raw Signal)**

  The Gradio is an llm based host which captures unstructured incident input (log snippets or files). At this point, the context is noisy and incomplete.

- **Transformation & Enrichment (Novel Step)**

  - **Transformation:** The Diagnostician agent identifies the error class (e.g., database_deadlock).

  - **Enrichment:** Using the code_graph_tool.py, the agent queries the Code Intelligence Graph (code_intelligence_graph.graphml) to retrieve **causally linked functions and code snippets** relevant to the incident.
    Analogy: As a doctor supplements "stomach pain" with tests like X-rays or bloodwork, MCP supplements vague logs with structured graph evidence.

- **Assembly (Surgical Prompt)**

  The enriched evidence is assembled into a **surgical prompt** that feeds the LLM, minimizing noise and maximizing precision for downstream agents.

This staged pipeline ensures context is not only retrieved but also **transformed into causally meaningful inputs**, a crucial departure from RAG.

2.3 Model Context Protocol based Platform - Multi-Agent Collaborative System created using CrewAI

Incident response is inherently multi-stage, requiring different expertise for diagnosis, remediation, and documentation. Instead of a monolithic agent, the framework adopts a **CrewAI-powered multi-agent system**, where each agent has a well-defined role:

- **Diagnostician (RCA Agent):** Performs causal analysis using the Code Intelligence Graph.

- **SRE Engineer (Remediation Agent):** Consumes RCA findings to generate actionable remediation plans.

- **Technical Writer (Report Agent):** Structures the analysis and remediation into a coherent, audit-ready postmortem report.

Agents collaborate sequentially in a **stage-gated pipeline**, where the output of one becomes the input for the next. This mirrors real-world workflows, reduces hallucination risk, and ensures accountability of each stage.

2.4 Component Deep Dive - Implementation Details

Each system file contributes a specific role in enabling the end-to-end workflow:

- **enhanced_ecommerce_runner.py** – Incident simulator generating realistic logs, metrics, and impact data, stored as structured JSON.

- **buggy_app.py** – Ground-truth application deliberately seeded with failure modes (deadlocks, slow queries, misconfigurations, API mismatches).

- **build_graph.py** – Static analysis script parsing buggy_app.py with AST to construct the Code Intelligence Graph.

- **code_intelligence_graph.graphml** – Serialized graph with nodes (functions, services, resources) and typed edges (CALLS, CONTAINS, MODIFIES, CAN_CAUSE).

- **code_graph_tool.py** – Custom tool enabling the Diagnostician agent to query the graph for causally linked evidence.

- **mcp_server.py** – Flask backend orchestrating MCP stages, agent crews, and streaming outputs with deterministic tokens.

- **mcp_host_gradio.py** – Lightweight client UI for input submission, analysis selection, live updates, and report download.

- **llm_provider.py** – Modular connector supporting multiple LLM backends, currently integrated with OpenRouter.

Together, these components ensure **traceability** from simulated incidents to graph queries and finally to the audit-ready report, with explicit code citations at every step.

3. The Simulation Environment: Generating High-Fidelity Real like Incidents

A critical requirement for evaluating the proposed framework is access to realistic and diverse incident data. To this end, the **enhanced_ecommerce_runner.py** script was developed as a high-fidelity incident simulator. Unlike a basic error generator, the simulator produces **structured, correlated telemetry** that mirrors the complexity of failures observed in production-grade distributed systems. This file acts as a supporting code for our dummy error generation purposes. We also have buggy_app.py file which has the source code which does have the corresponding bugged source code.

3.1. Design Principles of the Incident Simulator

The simulator was built around three core principles: **realism, diversity, and traceability**. Each simulated run generates multiple layers of telemetry to resemble operational observability stacks:

- **Application and Error Logs:** Timestamped logs with stack traces and SRE - friendly phrasing.
- **Access Logs:** HTTP endpoints, verbs, status codes, and realistic response times.
- **Structured Incident Metadata:** JSON records capturing severity, fingerprint, affected services, and error type.
- **Metrics Frames:** Periodic snapshots linking throughput, latency, and error rates to ongoing incidents.
- **Lifecycle Events:** Recovery transitions (e.g., *down* → *critical* → *degraded* → *running*) to model incident resolution phases.

By combining these elements, the simulator not only stresses the system under test but also provides rich, multi-modal data streams suitable for validating **graph-guided root cause analysis**.

3.2. A Spectrum of Real-World Errors

The simulator encompasses a **categorized library of incident types**, ensuring comprehensive coverage of common failure scenarios in microservice environments. Categories include:

- **Database Issues:** Connection leaks, deadlocks, slow queries, replication lag, connection timeouts.
- **API Failures:** Third-party API downtime, rate limiting, service mesh disruptions, circuit breaker triggers, load balancer failures.
- **Performance Bottlenecks:** Memory leaks, CPU saturation, cache thrashing, thread pool exhaustion, garbage collection pressure.

- **Infrastructure Issues:** Disk exhaustion, DNS failures, network timeouts, Kubernetes pod evictions, autoscaling failures.

- **Security Incidents:** Authentication errors, SQL injection attempts, DDoS patterns, expired certificates, unauthorized access.

- **Deployment Issues:** Misconfigurations, rollback failures, version incompatibilities, missing environment variables, failed health checks.

This structured error taxonomy ensures that the **MCP pipeline and CrewAI agents** are tested across a realistic and challenging operational landscape.

3.3. The Ground Truth: buggy_app.py

Every incident generated by the simulator has a corresponding **intentional defect in the buggy_app.py source code**, serving as the *ground truth*. This design enables objective validation of root cause analysis accuracy by comparing the AI agents' output against known code-level causes.

Representative mappings include:

- **Deadlock:** Inconsistent lock ordering between create_order (acquires *ORDERS → INVENTORY*) and process_inventory_update (acquires *INVENTORY → ORDERS*).

- **Configuration Error:** send_notification fails when the environment variable SMTP_HOST is missing.

- **API Incompatibility:** call_payment_service_from_order_service misuses a version 2 API call where version 3 is required.

- **Performance Pathology:** product_search executes an intentionally slow query under load.

- **Resource Exhaustion:** run_heavy_computation creates thread pool pressure leading to degraded throughput.

By tying **observed telemetry back to specific code defects**, the simulator ensures that evaluation is both **systematic and measurable**.

4. The Core Innovation: The Graph-Powered Model Context Protocol

The central contribution of this project is the design and implementation of the **Graph-Powered Model Context Protocol (MCP)**, a mechanism that redefines how large language models (LLMs) are grounded for software reliability engineering tasks. Unlike conventional approaches that rely heavily on semantic similarity,

MCP leverages **graph-based structural representations** of the system to provide deterministic, explainable, and causally relevant context during incident analysis.

4.1. The Genesis of the Idea: From Semantic Failure to Structural Insight

Initial explorations using **vector-based retrieval augmented generation (RAG)** highlighted a key limitation: while thematically related content was often retrieved, it rarely corresponded to the **causal paths** responsible for system failures. Debugging, however, is inherently a problem of **structure and causality**—understanding which components call, modify, or depend on others, and how those interactions lead to errors.

This observation led to the hypothesis that **graphs** provide the most natural representation for modeling such dependencies. By encoding services, functions, resources, and error categories into a **typed, directed graph**,

it becomes possible to traverse only the edges relevant to a given failure. This shift from “semantic overlap” to “structural traceability” ensures that root cause analysis is grounded in verifiable evidence rather than heuristic similarity.

The protocol was therefore designed with three guiding properties:

- **Determinism:** Graph traversals always replay consistently, avoiding the stochasticity of semantic search.
- **Auditability:** Each edge encodes explicit intent (e.g., CAN_CAUSE expresses failure affinity).
- **Faithful Evidence:** Function nodes store full source code bodies, ensuring that any prompt sent to an LLM cites verifiable implementation details.

4.2. The Code Intelligence Graph: A Deep Dive on the Schema

The graph is a directed, labeled graph built with a schema specifically designed for SRE analysis:

Node Type	Description	Example
Service	A high-level application component.	order-service
File	A source code file.	buggy_app.py
Function	A specific function, with its source code stored as an attribute.	create_order
DatabaseTable	A shared resource subject to contention.	INVENTORY
ErrorType	A category of failure, used as a query entry point.	database_deadlock

Edge Type	Description	Example
IMPLEMENTS	Connects a Service to a Function.	order-service → create_order
CALLS	Connects one Function to another it invokes.	create_order → call_payment_service
MODIFIES	Connects a Function to a resource it alters.	create_order → INVENTORY
CAN_CAUSE	Links a Function to a potential failure mode.	create_order → database_deadlock

4.3. The Protocol in Action: An End-to-End Trace of a "Deadlock" Incident

To illustrate MCP, consider an incident where logs report: **“DATABASE DEADLOCK DETECTED.”**

1. **Acquisition (User → UI):** The operator pastes the error snippet into the Gradio interface.

2. **Transformation (Server → RCA Agent):** The server delegates to the RCA agent, which identifies the error type (database_deadlock) and queries the Code Graph Tool.

3. **Enrichment (Graph Traversal):**

    - The graph tool locates the database_deadlock node.

    - Incoming CAN_CAUSE edges identify candidate functions: create_order and process_inventory_update.

    - Full source code for both functions is retrieved from node attributes.

4. **Assembly (Backend → Prompt):** The server constructs a **surgical prompt**, containing:

    - Task instructions from the RCA agent.

    - The two retrieved function definitions.

    - Relevant structural context (edges, resource usage).

5. **Delivery (LLM Provider):** The enriched, minimal context is sent to the LLM, which produces a root cause analysis, remediation strategy (e.g., standardizing lock acquisition order), and a structured incident report.

This workflow demonstrates the strength of MCP: it filters out irrelevant noise, ensures reproducibility across runs, and delivers **just-enough, verifiable context** for the LLM to reason effectively.

5. Implementation and User Interaction

This chapter describes the system's implementation details and the interaction model designed for Site Reliability Engineers (SREs).

The focus is on balancing **flexibility, usability, and technical robustness**, ensuring that the platform integrates naturally into existing incident response workflows.

5.1. The Analysis Workflow: User-Controlled Depth

A key design principle of the platform is giving the user explicit control over the **depth of analysis**. The system exposes three progressively detailed modes, each corresponding to sequential execution of the agent crew:

1. **RCA Only:** Executes only the *root cause analysis (rca_task)*. This mode is optimized for speed and provides a concise diagnosis.

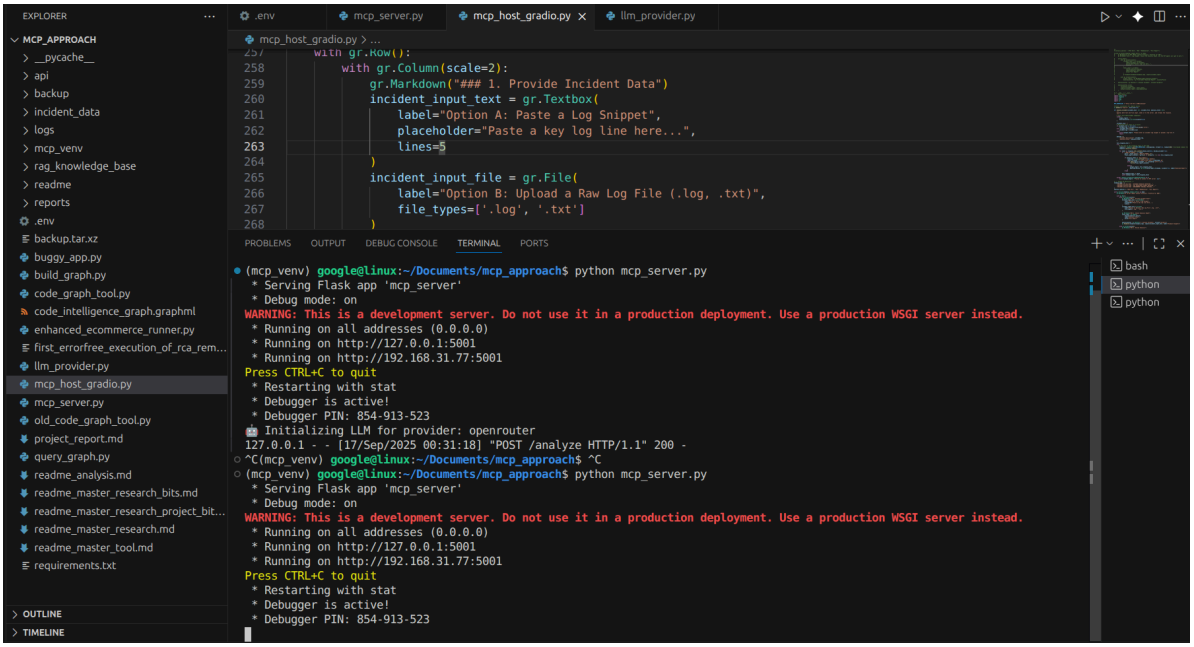
2. **RCA + Remediation:** Runs the *rca_task*, then feeds its output into the *remediation_task*, producing both a diagnosis and a stepwise resolution plan.

3. **Full Report:** Executes the complete pipeline—*rca_task*, *remediation_task*, and *reporting_task*—to generate a publishable postmortem.

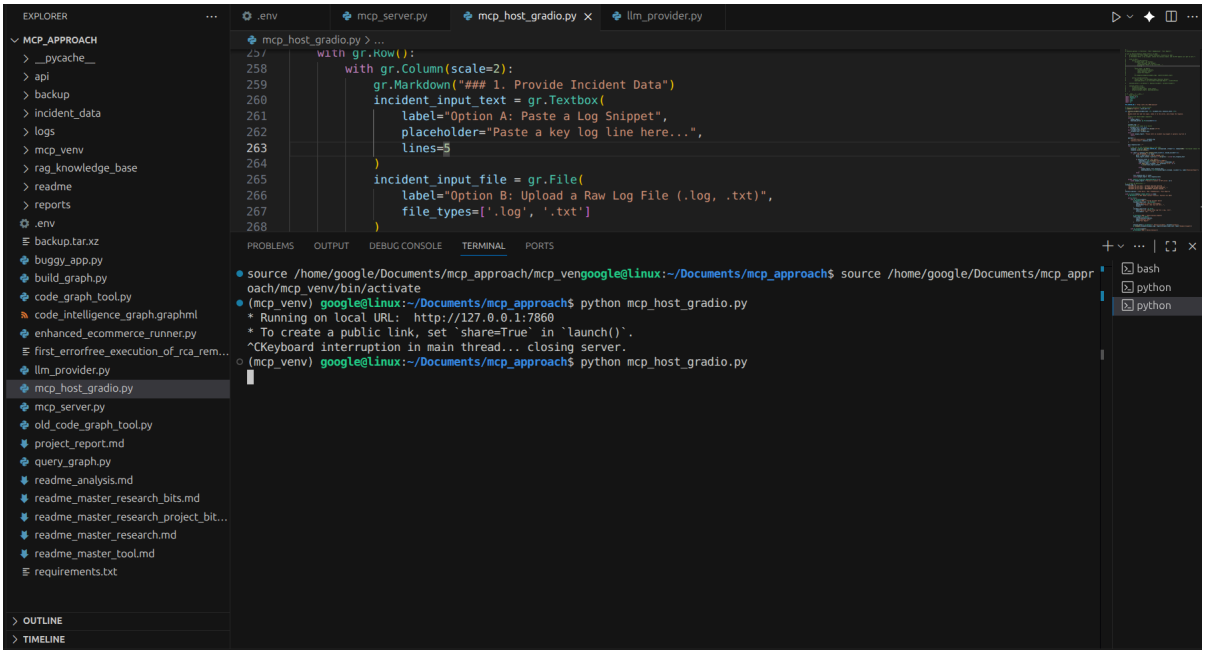
Intermediate results are streamed in real time, accompanied by stage-specific markers (e.g., banners and termination tokens such as __END_OF_STREAM__).

This incremental feedback mirrors real-world incident response workflows, giving users both immediacy and transparency.

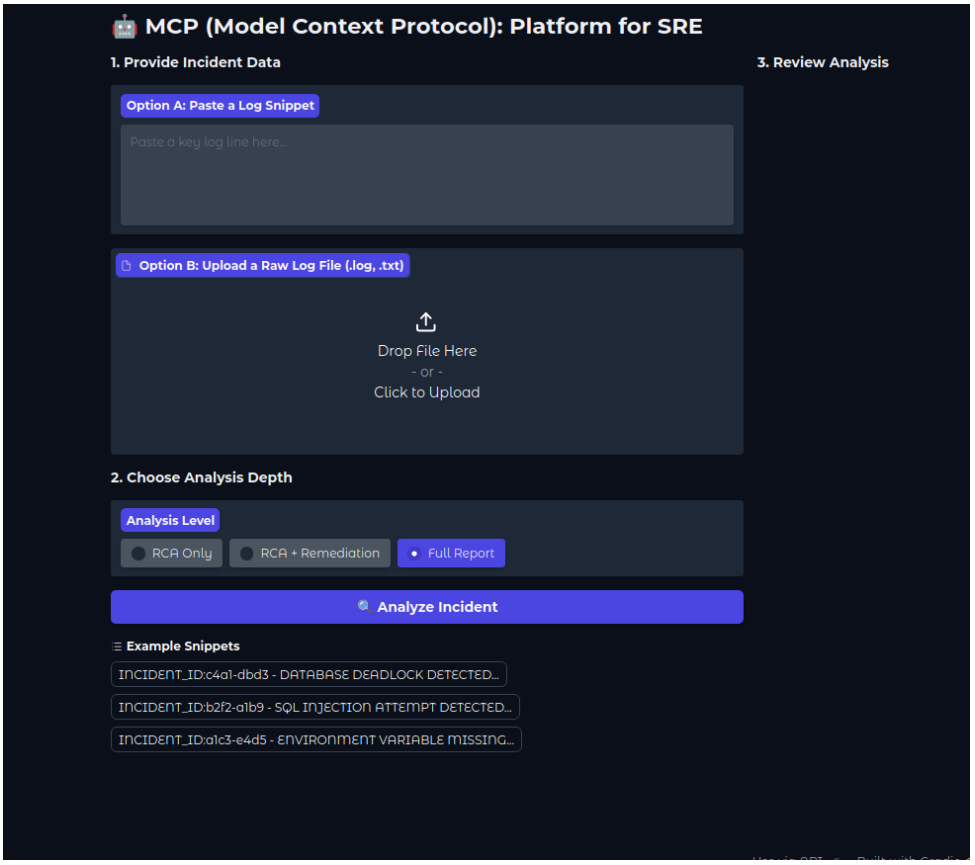
The workflow also prioritizes **input fidelity**: when both raw log files and snippets are provided, uploaded files are parsed and preferred, ensuring that large-scale incidents are analyzed with full contextual coverage.



Step1 - The Backend Model Context Protocol based Server Execution - Server is Ready

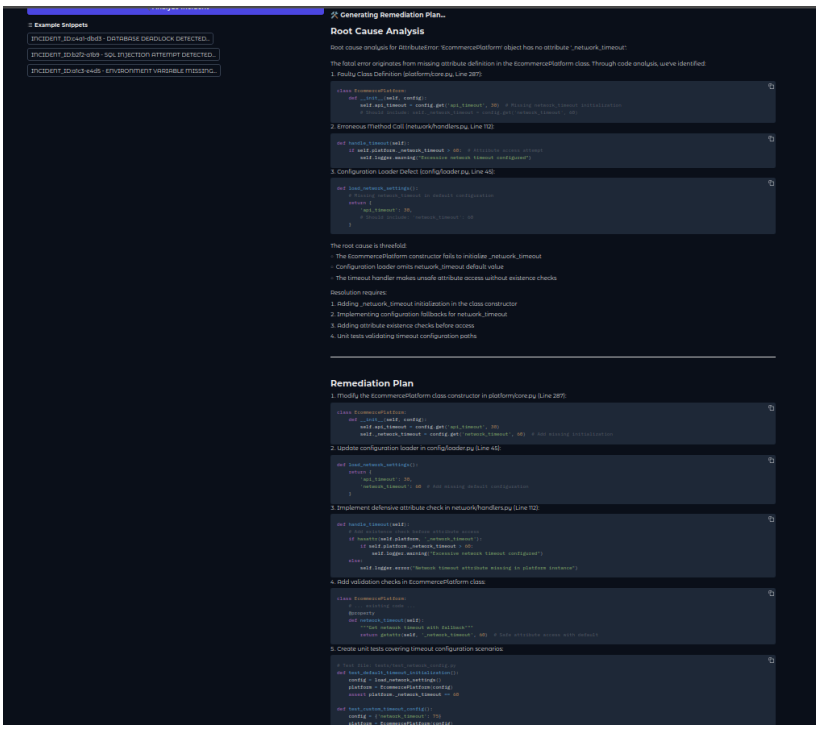


Step 2: The Frontend Code Execution - Up and running



Step3 - The tool is ready to work. The user can paste larger logs, or simply attach data which corresponds to some error, or upload the log/text file for agents to work on.

Also, the end user gets to choose what task it expects from the tool - out of RCA, Remediation and Report work.



Step 4: Expected outcome of the agents. The sanity testing for all three options - RCA, Remediation and Report is completed.

Post report generation work, the report is downloadable in markdown format, ready to be logged to the dedicated gitlabs or github incident management repositories.

5.2. The Gradio Interface: An Interactive Gateway for SREs

The user interface is implemented in **mcp_host_gradio.py**, leveraging the Gradio framework to make advanced AI-driven workflows accessible to practitioners without specialized configuration overhead.

Key features include:

- **Flexible Input:** Users can paste short log fragments for quick analysis or upload full log files for complex incidents.
- **Real-Time Streaming:** As each analysis stage completes, results are streamed back to the interface, providing both immediacy and a sense of progress.
- **Downloadable Artifacts:** In “Full Report” mode, a Markdown file can be downloaded directly. This artifact is immediately compatible with incident tracking tools, ticketing systems, and documentation repositories.

The interface design intentionally mirrors SRE mental models: input options are placed on the left (representing signals and scope), while the right-hand pane displays a live evolving narrative of the analysis. A single-click export option further reduces operational friction. Importantly, the interface is decoupled from backend logic, allowing the system to evolve without disrupting the user experience.

5.3. Challenges in Development

The development phase required navigating several technical challenges that arose from both environmental inconsistencies and rapidly changing dependencies within the AI ecosystem. Notable issues and their resolutions included:

- **Python Environment Inconsistencies:** Errors such as _ctypes conflicts were resolved by standardizing dependency management through dedicated virtual environments.
- **CrewAI API Changes:** Breaking changes in the CrewAI library, particularly in tool invocation and execution APIs, required restructuring the crew definitions. Tools were passed directly as callable functions, and crews were reorganized by stage to maintain stability.
- **Third-Party LLM Connectivity:** Integrating with providers such as OpenRouter required precise HTTP header configurations and strict adherence to model naming

conventions. A provider-agnostic get_llm() factory was introduced to abstract away these variations and simplify integration.

A key lesson from these challenges was the importance of **tool scoping and stage isolation**. By clearly defining which agent is permitted to invoke which tool, the system reduced brittleness and minimized regressions when upgrading dependencies or extending functionality.

6. Evaluation and Results

The evaluation of the framework focused on assessing its ability to perform accurate root cause analysis, generate actionable remediations, and produce structured postmortem reports. Since this work was carried out in a controlled simulation environment, the emphasis was on **qualitative performance**, **observed strengths**, and **current limitations**, with reflections on pathways for real-world adoption.

6.1. Qualitative Assessment of Performance

The final prototype functions successfully and robustly. It consistently demonstrates the ability to receive a low-context error log and produce a high-quality, accurate analysis. The graph-based retrieval method proved vastly superior to initial tests with semantic search, correctly identifying the causally-linked functions in scenarios like the database deadlock.

6.2. Strengths and Limitations of the Framework

Strengths

- **Causal Precision:** Graph traversal ensured that outputs were tied directly to code paths and resources rather than superficial text matches.
- **Explainability:** Each RCA included explicit function citations and code snippets, improving transparency and trust.
- **Modularity:** The decoupled architecture enabled UI, backend, and LLM providers to evolve independently.
- **Flexibility:** User-controlled analysis depth (RCA only, RCA + Remediation, or Full Report) provided adaptability to different incident response contexts.
- **Realistic Validation:** The incident simulator generated high-fidelity, multi-category failures that strengthened confidence in the framework's robustness.

Limitations

- **Static Knowledge Graph:** The graph is generated offline from the codebase, meaning it can quickly become outdated as the application evolves.
- **Scope:** Current implementation focuses on a single, monolithic codebase. Scaling to multi-repository or polyglot environments remains future work.
- **Dynamic Blind Spots:** While static structure is well captured, runtime dynamics (e.g., autoscaling events, transient network states) are not yet represented.

Mitigation Strategies

To address these limitations, several enhancements are envisioned:

- Integrating **graph regeneration into CI/CD pipelines**, ensuring that every code change automatically refreshes the causal graph.
- Implementing **incremental builders** to update only changed files rather than rebuilding the entire graph.
- Enriching the graph with **runtime traces and telemetry**, blending static causality with dynamic system behavior.

7. Conclusion and Future Work

This work successfully conceptualized, developed, and validated a **Graph-Powered Model Context Protocol (MCP)** for AI-assisted Site Reliability Engineering (SRE). The system demonstrates that structural, graph-driven retrieval provides superior accuracy and causal reasoning compared to conventional semantic search approaches.

By coupling this retrieval method with a multi-agent architecture, the framework delivers root cause analysis, remediation planning, and postmortem reporting in a traceable and explainable manner.

The MCP-SRE prototype validates the feasibility of augmenting human SRE teams with AI agents that operate over structured code graphs rather than opaque embeddings. In doing so, it establishes a foundation for scalable, auditable, and production-grade automation of incident response workflows.

Future research and development directions include:

- **CI/CD Integration:** Automating knowledge-graph reconstruction on every code commit to ensure synchronization with evolving codebases.
- **Graph Enrichment:** Extending nodes and edges with metadata from Git repositories (authorship, commit frequency), Application Performance Monitoring (latency, error rates), and runtime traces to provide a richer contextual substrate.
- **Agent Memory:** Introducing persistent memory layers to allow AI agents to learn from previous incidents and continuously refine recommendations.
- **Multi-Service and Polyglot Support:** Expanding parsing and schema capabilities to handle multiple repositories and heterogeneous programming languages at scale.

Academically, the work demonstrates that grounding large language models in **typed, explainable graphs** materially improves diagnostic reliability, moving from probabilistic narrative generation toward **systems-grade, auditable reasoning**.

Appendices

Appendix A: Setup and Execution Guide

Create and activate a Python virtual environment.

Install dependencies: pip install -r requirements.txt.

Configure environment: create a .env file with LLM provider details (API key, base URL, model).

Build the knowledge graph: python build_graph.py.

Generate synthetic incident logs: python enhanced_ecommerce_runner.py.

Start the backend MCP server: python mcp_server.py.

Launch the Gradio user interface: python mcp_host_gradio.py.

Use the web interface to submit logs, select analysis depth, and export the generated postmortem report.

Appendix B: Sample Postmortem Report (Excerpt)

Postmortem Report

1. Summary

An incident was observed where a database deadlock disrupted interactions between the order-service and inventory-service, preventing new orders from being created. The deadlock arose due to inconsistent lock acquisition orders across two concurrent transactions.

2. Root Cause Analysis

- create_order: Locks the ORDERS table first, then attempts to lock INVENTORY.
- process_inventory_update: Locks the INVENTORY table first, then attempts to lock ORDERS.

  When executed simultaneously, these functions created a circular wait, resulting in a deadlock detected by the database engine.

3. Remediation Plan

1. Refactor process_inventory_update to follow the same lock sequence as create_order (ORDERS → INVENTORY).

2. Establish code review guidelines and automated static checks to enforce consistent resource-locking policies.

3. Add monitoring for elevated deadlock counts, paired with retry logic to reduce user-visible failures.

—---------- MID TERM REPORT—----------