BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
WORK INTEGRATED LEARNING PROGRAMMES DIVISION
BITS WIPRO COLLABORATIVE PROGRAMME

M Tech in Computing Systems & Infrastructure

8^{th} SEMESTER 2025- 26

CSIWZG628T DISSERTATION

AI-Powered Diagnostics & Failure Analysis Engine for Jenkins with MCP

Vaibhav Karad
2021WC86419

Supervisor:
Krishan Kant Sahu
Software Architect

Examiners:
Rohit Dagur
DevOps Lead

Simone Bruno
DevOps Principal Consultant

November 2, 2025

Abstract

The research and development focus on creating an intelligent, scalable, and centralized diagnostics system for Jenkins CI/CD pipelines. The core area involves applying Artificial Intelligence (AI), specifically Large Language Models (LLMs) and vector-based search, to automate the analysis of build failures, manage massive log files, and orchestrate multiple Jenkins instances from a central Master Control Plane (MCP).

The primary purpose of this project is to develop a scalable, AI-enhanced Jenkins MCP server that streamlines build failure diagnosis, manages multiple Jenkins instances centrally, and delivers actionable insights in real time. The system aims to significantly reduce the Mean Time to Repair (MTTR), improve overall pipeline stability, and enable efficient, intelligent debugging across enterprise-scale CI/CD environments.

This report details the project's background, methodology, and complete system architecture. It provides a comprehensive guide to the implementation, setup, and usage of the "Jenkins AI Optimizer," including its integration with Claude Desktop. Finally, it outlines the monitoring, maintenance, and troubleshooting procedures for the entire application stack.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background of Previous Work

As organizations increasingly adopt microservices, DevOps principles, and hybrid cloud infrastructures, the size and complexity of their CI/CD pipelines have grown exponentially. Jenkins, while a dominant force in CI/CD automation, presents significant challenges in these large-scale environments. Traditional Jenkins setups lack intelligent tooling for diagnosing build failures, efficiently managing a fleet of Jenkins instances, or processing terabytes of log data effectively.

Organizations frequently struggle with recurring failure patterns, time-consuming root cause analysis, and siloed build data that is fragmented across various teams and servers. These operational inefficiencies demand a smarter, AI-driven approach to diagnostics and a more robust, centralized management system. This project, `jenkins-ai-optimizer`, is founded on addressing these specific gaps in the current CI/CD landscape.

1.2 Objective and Scope of Work

1.2.1 Objective

The primary purpose of this project is to develop a scalable, AI-enhanced Jenkins MCP server that streamlines build failure diagnosis, manages multiple Jenkins instances centrally, and delivers actionable insights in real time. The system aims to significantly reduce the Mean Time to Repair (MTTR), improve overall pipeline stability, and enable efficient, intelligent debugging across enterprise-scale CI/CD environments.

1.2.2 Scope of Work

The project involves the following key deliverables:

- **AI-Powered Diagnostics Engine:** Implement LLM-based error interpreters and a pattern recognition system that learns from past failures to predict and diagnose issues.
- **Hierarchical Sub-Build Navigator:** Develop a recursive pipeline tree builder with a drill-down UI or API endpoints for clear visualization and navigation of complex builds.

- **Massive Log Handler:** Engineer a system to support streaming, chunking, and real-time analysis of large log files (exceeding 10GB).
- **Vector Search Integration:** Embed log data into semantic vectors and implement a high-speed similarity search using a vector database to find related failures.
- **Enterprise Multi-Jenkins Orchestration:** Build a central MCP featuring an instance registry, health monitoring capabilities, and logic for automatic job routing and failover.
- **Authentication & Security:** Incorporate SSL support and secure, per-instance credential management.
- **Performance & Scalability Optimization:** Implement parallel processing, strategic caching, and HTTP streaming to ensure real-time diagnostic performance.
- **Documentation & Deployment:** Provide complete system documentation, user guidelines, and a containerized deployment solution using Docker.

1.3 Glossary of Terms

MCP Master Control Plane. The central backend server that orchestrates communication and tooling.

Vector Store The Qdrant database used for storing log embeddings.

Tool A discrete executable capability exposed by the MCP server to the client (e.g., `diagnose_build_fa`

Embedding A numerical representation (vector) of text, used for semantic similarity search.

1.4 Report Structure

This report is organized into the following chapters:

- **Chapter 1: Introduction** provides the project background, objectives, scope, and glossary.
- **Chapter 2: System Architecture & Methodology** details the project's design, core components, repository structure, CI/CD pipeline, and security model.
- **Chapter 3: Implementation & Setup Guide** offers a step-by-step guide to installing, configuring, and starting the system, including a contributor setup.
- **Chapter 4: System Usage & User Guide** explains how to interact with the system using Claude Desktop, with detailed command examples and an FAQ.
- **Chapter 5: Monitoring & Maintenance** covers system monitoring, log locations, troubleshooting, and data management.
- **Chapter 6: Project Management & Conclusion** outlines the project plan, development workflow, and release process, concluding the report and suggesting future work.

Chapter 2

System Architecture & Methodology

2.1 Methodology

The project was executed using a combination of software development, systems integration, and machine learning methodologies:

- **System Architecture Design:** A microservices-based architecture was designed for the central Master Control Plane (MCP) to ensure scalability and modularity. The system comprises three main services: the MCP Server, the Qdrant Vector Database, and the Claude Desktop client.
- **AI Model Integration:** A pre-trained Large Language Model (LLM) is leveraged via the Claude Desktop interface. This model is responsible for interpreting user queries and the JSON-based tool outputs from the MCP server.
- **Vector Database Implementation:** Log data is processed, chunked, and converted into numerical embeddings using the `all-MiniLM-L6-v2` model. These vectors are stored in and indexed by a Qdrant vector database to enable rapid, semantic searching.
- **Backend Development:** The MCP server and associated APIs were developed in Python, managing Jenkins instances, handling API requests, and serving diagnostic data.
- **Containerization and Deployment:** The entire application stack (MCP Server and Qdrant) is containerized using Docker and orchestrated with `docker-compose` to ensure portability and ease of deployment.
- **Testing:** The project underwent unit, integration, and functional testing to validate reliability and performance.

2.2 Core System Architecture

The Jenkins AI Optimizer is an AI-powered assistant that connects to a Jenkins server to diagnose build failures, analyze pipelines, search logs, and trigger builds. The system works with any existing Jenkins job without requiring modifications or plugins.

The architecture consists of four primary components:

1. **Jenkins Server:** The user's existing CI/CD server. The MCP server communicates with it via its standard REST API.

2. **MCP Server:** The ”Master Control Plane” backend. This Docker container runs a Python server that exposes tools for Claude Desktop to consume. It handles all logic for fetching logs, analyzing failures, and querying the vector database.

3. **Qdrant Vector Database:** A Docker container running the Qdrant vector database. It stores embedded Jenkins console logs to power semantic (natural language) search.

4. **Claude Desktop:** The user interface. The user interacts with Claude in natural language. Claude interprets the request and uses the ”tools” provided by the MCP server to perform actions and return results.

System Architecture Diagram
(User → Claude Desktop → MCP Server → [Jenkins | Qdrant])

Figure 2.1: High-level system architecture and data flow.

2.3 Service Endpoints & Access

The system exposes several key endpoints for operation and monitoring. All services, except for the existing Jenkins server, are bound to `localhost` for security.

Table 2.1: Service URLs and Endpoints

Service	URL	Description
Jenkins Server	`http://172.26.128.1:8080`	The main Jenkins instance
MCP Server	`http://localhost:3008/health`	Health check for the MCP
Qdrant Dashboard	`http://localhost:6333/dashboard`	Web UI for the vector database
Qdrant API	`http://localhost:6333`	REST API for the vector database

2.4 Repository Structure

The project source code is organized into a primary Python package, configuration, tests, and Docker definitions.

```
jenkins-ai-optimizer/├──
 jenkins_mcp_enterprise/      # Primary Python package│├──
    cli.py                   # CLI entrypoint│├──
    server.py / base.py│├──
    config.py│├──
    di_container.py          # Dependency wiring│├──
    diagnostic_config/       # Diagnostics schemas│├──
    jenkins/                 # Jenkins integration│├──
    tools/                   # Tool implementations│├──
    streaming/               # Log/event streaming│├──
    vector_manager.py        # VectorDB management│├──
    cache_manager.py│├──
    cleanup_manager.py│├──
    multi_jenkins_manager.py│└──
    exceptions.py├──
 config/                      # Example configs├──
 scripts/                     # Helper scripts├──
 tests/                       # Pytest suite├──
 Dockerfile / docker-compose.yml├──
 pyproject.toml / requirements.txt└──
 README.md
```

Listing 2.1: Repository File Structure

2.5 Vector Database (Qdrant)

The Qdrant vector database is a critical component for enabling semantic log search.

2.5.1 Data Collection Flow

When a user asks Claude to diagnose a build, the following data flow is initiated:

1. **Trigger:** User asks Claude to diagnose a build (e.g., "`Diagnose build #1`").
2. **Fetch:** The MCP server downloads the full console log from the Jenkins API.
3. **Cache:** The raw log is saved to a temporary cache inside the MCP Docker container at `/tmp/mcp-jenkins/`.
4. **Chunk:** The log is split into semantic chunks (e.g., ~200 lines each).
5. **Embed:** Each chunk is converted into a 384-dimensional vector using the `all-MiniLM-L6-v2` embedding model.
6. **Store:** The vector and its associated metadata (payload) are stored in the Qdrant database within the `jenkins-logs` collection.
7. **Index:** The new vector is indexed and becomes available for semantic search queries.

2.5.2 Data Payload Structure

Each vector stored in Qdrant is associated with a JSON payload containing metadata about the log chunk. This allows the system to retrieve contextually relevant information.

```json
{
  "job_name": "Java_System_Info",
  "build_number": 1,
  "chunk_text": "ERROR: Connection timeout...",
  "line_start": 145,
  "line_end": 160,
  "timestamp": "2025-10-30T09:03:48Z",
  "jenkins_url": "http://172.26.128.1:8080",
  "chunk_index": 2,
  "total_chunks": 8
}
```

Listing 2.2: Qdrant vector payload structure

2.6 CI/CD Pipeline & Quality

The project maintains quality through an automated CI/CD pipeline with defined stages and quality gates.

2.6.1 Conceptual CI Stages

Table 2.2: CI/CD Pipeline Stages

Stage	Purpose
Setup	Install dependencies, restore cache
Lint & Format	Enforce style (Ruff, Black)
Unit Tests	Fast feedback (pytest)
Integration Tests	System behavior (docker-compose, Jenkins stub)
Security Scan	Dependencies & image (pip-audit, trivy)
Build Artifacts	Package wheels / sdist
Docker Image	Runtime packaging (Multi-stage build)
Publish	Publish to PyPI / registry (on tag only)

2.6.2 Quality Gates

An item is considered ”Done” only when it meets these criteria:

- No new linting errors; warnings triaged.
- All tests pass (e.g., >75% coverage).
- No plaintext secrets committed.
- No high/critical vulnerable dependencies.
- Documentation updated to reflect changes.

2.7 Security & Compliance

Security is enforced through several mechanisms:

- **Dependency Scanning:** Automated weekly scans for vulnerable packages.

- **Access Control:** Direct commits to the `main` branch are prohibited; pull requests with at least one reviewer are required.
- **Secret Scanning:** The repository is monitored for accidental commitment of plaintext secrets.

Chapter 3

Implementation & Setup Guide

3.1 System Startup (User)

The backend services (MCP Server and Qdrant) are managed via `docker-compose`.

1. Navigate to the project's root directory:

```
cd C:\Users\Vaibhav\Documents\GitHub\jenkins-ai-optimizer
```

Listing 3.1: Navigate to project directory

2. Start the Docker containers in detached mode:

```
docker-compose up -d
```

Listing 3.2: Start Docker containers

3. Verify that both containers are running and healthy:

```
docker ps
```

Listing 3.3: Verify container status

The expected output should show `jenkins_mcp_enterprise-server` and `jenkins_mcp_enter` `qdrant` with a ”healthy” status.

4. Perform a health check on the MCP server:

```
curl http://localhost:3008/health
# Expected response: "OK"
```

Listing 3.4: MCP server health check

3.2 Jenkins Instance Configuration

The MCP server must be configured to connect to the Jenkins instance. This is done in the `config/mcp-config.yml` file.

```
jenkins_instances:
  production: # Logical name for the instance
    url: "http://172.26.128.1:8080" # Your Jenkins URL
    username: "your-username"
    token: "your-api-token" # Jenkins API Token
    timeout: 30

default_instance: "production"

vector:
  enabled: true
  host: "http://qdrant:6333" # Internal Docker network hostname
  collection_name: "jenkins_logs"
  embedding_model: "all-MiniLM-L6-v2"
```

Listing 3.5: Jenkins instance configuration (`mcp-config.yml`)

To obtain a Jenkins API Token:

1. Log into the Jenkins web interface.
2. Click your username (top right) and select ”Configure”.
3. Navigate to the ”API Token” section.
4. Click ”Add new Token”, provide a name, and copy the generated token.

Note: This system requires **zero changes** to existing Jenkins jobs. It does not require any special plugins, Jenkinsfile modifications, or configuration changes on the Jenkins server itself. It operates entirely via the standard Jenkins REST API.

3.3 Claude Desktop Integration

To connect the Claude Desktop application to the local MCP server, modify the Claude configuration file.

1. Locate the configuration file at: `C:\Users\Vaibhav\AppData\Roaming\Claude\claude_desk`
2. Add the following JSON object to enable the `jenkins` MCP server:

```
{
  "mcpServers": {
    "jenkins": {
      "url": "http://localhost:3008/sse"
    }
  }
}
```

Listing 3.6: Claude Desktop configuration (`claude_desktop_config.json`)

3. Completely close and restart the Claude Desktop application for the changes to take effect. A ”plug” icon should appear, indicating a successful MCP connection.

3.4 Diagnostic Pattern Configuration

The AI’s ability to categorize errors is defined by customizable patterns in `jenkins_mcp_enterprise/` `parameters.yml`. This allows the system to be tailored to specific error types.

```
error_patterns:
  compilation:
    - pattern: "error: .* undeclared"
      severity: "high"
      category: "Compilation Error"

  timeout:
    - pattern: "timeout|timed out"
      severity: "medium"
      category: "Timeout"

  permission:
    - pattern: "permission denied|access denied"
      severity: "high"
      category: "Permission Error"
```

Listing 3.7: Example diagnostic patterns

3.5 Contributor Quick Start

For developers contributing to the project, the local setup involves a virtual environment and pre-commit hooks.

```
# Clone the repository
git clone https://github.com/<org>/jenkins-ai-optimizer.git
cd jenkins-ai-optimizer

# Create a virtual environment
python -m venv .venv
source .venv/bin/activate  # Or .venv\Scripts\activate

# Install dependencies in editable mode
pip install -e .[dev]

# Run linting and unit tests
ruff check .
pytest -m "not integration"
```

Listing 3.8: Contributor setup commands

Chapter 4

System Usage & User Guide

This chapter details how to interact with the Jenkins AI Optimizer using the Claude Desktop interface.

4.1 Prerequisite: Setting Jenkins URL

Before issuing commands, you must tell Claude which Jenkins server to target. This can be done once per session or with each command.

```
My Jenkins server is at http://172.26.128.1:8080
```

Listing 4.1: Setting the Jenkins URL

Alternatively, include it in every prompt:

```
Diagnose build #1 at http://172.26.128.1:8080
```

Listing 4.2: Including URL in prompt

4.2 Core Commands

The following sections outline the primary commands available through the natural language interface.

4.2.1 Diagnose Build Failures

This is the system's core function. It downloads, analyzes, and categorizes log failures, providing AI-powered recommendations.

- **Command:** `Diagnose the failure in [job-name] build #[number] at [jenkins-url]`
- **Examples:**

```
Diagnose the failure in Java_System_Info build #1 at http
    ://172.26.128.1:8080

Why did Windows_Health_Check build #2 fail at http://172.26.128.1:8080?
```

Listing 4.3: Build diagnosis examples

- **Action:** This triggers the diagnose_build_failure tool, which downloads the log, identifies error patterns, provides recommendations, and indexes the log in Qdrant for future semantic search.

4.2.2 Trigger New Builds

Users can start new Jenkins builds directly from the chat interface.

- **Command:** Trigger a build of [job-name] at [jenkins-url]
- **Examples:**

```
Run a new build of Java_System_Info at http://172.26.128.1:8080

Trigger Java_System_Info with parameters at http://172.26.128.1:8080:
- BRANCH: develop
- ENVIRONMENT: staging

Start Windows_Health_Check without waiting at http://172.26.128.1:8080
```

Listing 4.4: Trigger build examples

- **Action:** This triggers the trigger_build tool. By default, it waits for completion (10 min timeout). Using ”asynchronously” or ”without waiting” returns the queue ID immediately.

4.2.3 Log Search (Keyword)

Perform smart, keyword-based searches on cached logs.

- **Command:** Search for [pattern] in [job-name] build #[number] at [jenkins-url]
- **Examples:**

```
Find all errors in Java_System_Info build #1 at http
    ://172.26.128.1:8080

Search for "timeout" in Windows_Health_Check build #2 at http
    ://172.26.128.1:8080
```

Listing 4.5: Keyword search examples

- **Action:** Uses smart grep with relevance scoring to find and display log snippets with context.

4.2.4 Log Search (Semantic)

Perform AI-powered, natural language searches on indexed logs. This finds results based on *meaning* and *context*, not just keywords.

- **Command:** Semantically search for "[description]" in [job-name] build #[number] at [jenkins-url]
- **Examples:**

```
Find stack traces in Java_System_Info build #1 at http
    ://172.26.128.1:8080

Search for database connection issues in Windows_Health_Check #2 at
    http://172.26.128.1:8080
```

Listing 4.6: Semantic search examples

- **Action:** This triggers the `search_logs_semantic` tool, which converts the query into a vector and searches Qdrant for similar log chunks.

4.2.5 Log Navigation

Retrieve specific sections or line ranges from a log.

- **Command:** `Show lines [start]-[end] from [job-name] build #[number] at [jenkins-url]`
- **Examples:**

```
Show lines 100-200 from Java_System_Info build #1 at http
    ://172.26.128.1:8080

Get the first 50 lines of Windows_Health_Check #2 at http
    ://172.26.128.1:8080

Show the last 100 lines of Java_System_Info #1 at http
    ://172.26.128.1:8080
```

Listing 4.7: Log navigation examples

- **Action:** Fetches and displays specific line ranges from the cached log.

4.3 Example Conversation Flow

The following demonstrates a typical user workflow for diagnosing a failure.

You: My Jenkins is at `http://172.26.128.1:8080`

Claude: Got it! Your Jenkins server is noted. How can I help you today?

You: Diagnose the failure in Java_System_Info build #1

Claude: *[Uses diagnose_build_failure tool]*

I've analyzed Java_System_Info build #1. The build failed due to:

Error Category: Compilation Error

Key Issues:

- Line 145: `error: cannot find symbol: variable DATABASE_URL`
- Missing environment variable configuration

Recommendations:

1. Add `DATABASE_URL` to the job parameters or global configuration.

2. Check Jenkins credentials configuration if it’s stored as a secret.
3. Verify the environment setup script.

Would you like me to search for similar errors in other builds?

You: Yes, search for ”DATABASE_URL” errors

Claude: *[Uses ripgrep_search tool]*

Found 2 occurrences of ”DATABASE_URL” in the log for build #1:

- `[Build #1, Line 145] error: cannot find symbol: variable DATABASE_URL`
- `[Build #1, Line 287] Caused by: java.lang.NullPointerException: DATABASE_URL is null`

4.4 Frequently Asked Questions (FAQ)

Q: Where do I put a new diagnostic tool? **A:** New tools are added to the `jenkins_mcp_enterprise/` directory, following the existing pattern.

Q: How do I add a config parameter? **A:** Update the `config/` examples, adjust the schema in `diagnostic_config/`, and document the change in the README.

Q: Do I need to change my existing Jenkins jobs? **A:** No! The system works with any existing Jenkins job without modifications.

Q: Can I use this with multiple Jenkins servers? **A:** Yes. Add multiple entries to the `jenkins_instances` list in `mcp-config.yml`.

Chapter 5

Monitoring & Maintenance

This chapter covers the monitoring of system components, log locations, and maintenance procedures.

5.1 Accessing Dashboards

5.1.1 Qdrant Vector Database Dashboard

The Qdrant dashboard provides a web UI to inspect the state of the vector database.

- **URL:** `http://localhost:6333/dashboard`
- **Features:**
  - **Collections View:** Shows all vector collections. You will see the `jenkins-logs` collection here, along with its status (should be ”Green”) and the total number of vectors (log chunks) indexed.
  - **Collection Details:** Clicking `jenkins-logs` shows its schema, total points, and vector dimensions (384).
  - **Search Console:** Allows direct API-level testing of vector searches.
  - **Monitoring Metrics:** Displays memory usage, disk usage, and query performance.

5.1.2 Jenkins Server

The main Jenkins dashboard remains the primary source for build history and server status.

- **URL:** `http://172.26.128.1:8080`

5.2 Log Locations

Aggregated logs are crucial for debugging the optimizer system itself. They are separated by component.

To view the structure of the cached build logs inside the Docker container:

Table 5.1: System Log Locations

Component	Log Location / Command
MCP Server	`docker logs jenkins_mcp_enterprise-server -f`
Qdrant DB	`docker logs jenkins_mcp_enterprise-qdrant -f`
Claude Desktop	`$env:APPDATA\Claude\logs\mcp-server-jenkins.log`
Cached Build Logs	(Inside server container) `/tmp/mcp-jenkins/`

```
# List all cached builds
docker exec jenkins_mcp_enterprise-server ls -R /tmp/mcp-jenkins/

# Read a specific cached log
docker exec jenkins_mcp_enterprise-server cat /tmp/mcp-jenkins/instance-[
    UUID]/Java_System_Info/1/console.log
```

Listing 5.1: View cached log structure

5.3 Monitoring Commands

A set of shell commands can be used to quickly assess system health.

- **Check System Status:**

```
docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"
```

Listing 5.2: Check Docker process status

- **Check Resource Usage:**

```
docker stats --no-stream
```

Listing 5.3: Check Docker resource stats

- **Check Network Connectivity:**

```
# Test MCP server
curl http://localhost:3008/health

# Test Qdrant
curl http://localhost:6333/healthz

# Test Jenkins
curl http://172.26.128.1:8080/api/json
\end{stlisting}
\end{itemize}

\section{Data Cleanup}
\subsection{Automatic Cleanup}
The system is configured to automatically clean up old cached logs and
    vector data after 7 days. This is configurable in \texttt{config/
    mcp-config.yml}.

\subsection{Manual Cleanup}
To manually purge data:
\begin{itemize}
```

```
    \item \textbf{Clear all cached logs:}
\begin{lstlisting}[caption={Clear all cached logs}]
docker exec jenkins_mcp_enterprise-server rm -rf /tmp/mcp-jenkins/*
\end{DIF PRE-EDIT ADDED}
\end{DIF PRE-EDIT ADDED}
```

Listing 5.4: Check service health endpoints

- **Clear entire Qdrant collection:**

```
# Delete the collection
curl -X DELETE http://localhost:6333/collections/jenkins-logs

# Restart MCP server to recreate the collection
docker restart jenkins_mcp_enterprise-server
```

Listing 5.5: Clear Qdrant collection

5.4 Troubleshooting

- **Qdrant Dashboard Won't Load:** Check if the Qdrant container is running (`docker ps`). Check its logs (`docker logs ...`) and restart it if necessary (`docker restart ...`).
- **No Data in Qdrant:** This is expected on a fresh install. Data is only indexed *after* you ask Claude to diagnose a build for the first time.
- **MCP Server Not Responding:** Check its health (`curl http://localhost:3008/health`) and logs. A restart (`docker restart ...`) usually resolves connection issues.
- **Claude Desktop Not Connecting:** Ensure the MCP server is running. Verify the path and content of the `claude_desktop_config.json` file. Restart Claude Desktop completely.

Chapter 6

Project Management & Conclusion

6.1 Plan of Work

The project was executed over a 16-week period, following the phases outlined in Table 6.1.

Table 6.1: Project Plan of Work

Sr. No.	Phase	Activities	Timeline
1	Requirements Collection & Analysis	- Review all relevant documents - Analyze existing Jenkins functionalities - Understand business logic	End of Week 4
2	System Design	- Design system architecture - Define module interactions - Document UI/API expectations	End of Week 8
3	Implementation & Integration	- Develop core modules (AI diagnostics, log parser, multi-instance manager) - Integrate components	End of Week 13
4	Final Prep / Testing	- Perform functional and load testing - Fix bugs and optimize - Prepare deployment & go-live plan	End of Week 15
5	Project Report	- Compile all phases into final report - Include results and evaluation	End of Week 16

6.2 Project Workflow

Project work was organized using a Kanban-style GitHub Project board, following a consistent development workflow.

6.2.1 Development Workflow

1. **Triage:** New ideas are captured as Issues and triaged (labeled, prioritized).
2. **Backlog:** Accepted items are moved to the Backlog.

3. **In Progress:** A developer assigns the issue and creates a feature branch.
4. **In Review:** A Pull Request is opened, linking the issue. Automated CI checks (lint, test, scan) are executed.
5. **Done:** After review and approval, the PR is merged into `main` and the issue is automatically closed.

6.2.2 Branching & Naming

A trunk-based development model is used. Branch names follow a consistent pattern:

- `feature/<short-slug>` (e.g., `feature/multi-jenkins-routing`)
- `bugfix/<issue>-<slug>` (e.g., `bugfix/123-null-log-edge`)
- `refactor/<area>` (e.g., `refactor/tools-error-path`)

6.2.3 Pull Request & Review Process

Pull requests require a summary, linked issue, and a checklist verifying that tests, documentation, and security scans have been considered. Reviews prioritize correctness and clarity.

6.3 Release Process

Releases follow Semantic Versioning (MAJOR.MINOR.PATCH):

1. The version is updated in `pyproject.toml`.
2. A Git tag is created (e.g., `v0.5.0`).
3. Pushing the tag triggers the CI/CD pipeline to build and publish the package and Docker image.
4. Release notes are published on GitHub.

6.4 Continuous Improvement

The project health is tracked with lightweight metrics (e.g., lead time, review time, test flake rate). A rotating ”maintainer-of-the-week” is assigned to triage new issues and ensure CI health.

6.5 Conclusion

This project successfully demonstrates the development and implementation of a scalable, AI-enhanced Jenkins Master Control Plane. By integrating a ”bring-your-own-client” model with Claude Desktop, the system provides powerful, natural-language-driven tools for build diagnosis, log analysis, and pipeline orchestration.

The use of a vector database (Qdrant) for semantic log search moves beyond traditional keyword matching, enabling developers to find related issues based on contextual meaning. The containerized, API-first architecture ensures the system is scalable, maintainable, and easy to deploy.

The primary objective of streamlining build failure diagnosis and reducing Mean Time to Repair (MTTR) is met by providing immediate, AI-powered analysis and recommendations directly within the developer's chat interface.

6.6 Future Work

While the current system is robust, several areas from the project backlog offer potential for future enhancement:

- **Expanded CI/CD Support:** Adapt the MCP server to interface with other CI/CD platforms like GitLab CI, GitHub Actions, or CircleCI.
- **Proactive Failure Prediction:** Train a model on indexed log data to proactively identify builds that are likely to fail *before* they complete.
- **Automated Remediation:** Allow the AI to not only diagnose failures but also suggest and (with approval) attempt to apply fixes.
- **Dedicated Web UI:** Develop a full-featured web interface for the MCP, providing a centralized dashboard for managing all connected instances.
- **Observability:** Introduce OpenTelemetry instrumentation for distributed tracing.
- **Pluggable Backends:** Create a vector backend abstraction layer to support other databases (e.g., FAISS).

References / Bibliography

Bibliography

[1] Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional.

[2] Le, D. M., & Lo, D. (2020). DeepLog: A deep learning-based log anomaly detection and diagnosis system. *Journal of Systems and Software*, 168, 110651.

[3] Chen, B., et al. (2021). An Empirical Study on the Characteristics of Build Failures in the Maven Ecosystem. In *Proc. IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp 1-12.

[4] Jenkins Core committers. (2025). *Jenkins User Handbook*. Retrieved from `https://www.jenkins.io/doc/`

[5] Johnson, A. (2022). *Scalable Log Analysis using Vector Embeddings for Automated System Diagnosis*. Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, PA.