Privacy-Focused Agentic AI Workflow with Locally Deployable Language Models for Technical Debt Remediation in Software Systems

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Software engineering
July 2026
Abhishek Roy

Supervisors:
Ville Leppänen
Lauri Koivunen

UNIVERSITY OF TURKU
Department of Computing

Abhishek Roy: Privacy-Focused Agentic AI Workflow with Locally Deployable Language Models for Technical Debt Remediation in Software Systems

Master of Science (Tech) Thesis, 76 p., 11 app. p.
Software engineering
July 2026

Recent advances in large language models, particularly within agentic frameworks for software development and maintenance, have made it possible to work through a backlog of technical debt. However, the frontier models that can do this are third-party cloud services, which carry recurring API costs and raise concerns over data privacy and intellectual property. For organizations that are bounded by these constraints, alternative solutions are needed.

This thesis explores that alternative by designing, building and evaluating a privacy-focused multi-agent pipeline that remediates static analysis findings entirely on a locally served open-weight model. Following Design Science Research, the artifact integrates SonarQube for detection, tree-sitter for language-aware code editing, Git and the GitHub API for version control and Ollama for serving a relatively small Qwen3.5-27B language model. It delivers its fixes as a conventional, reviewable pull request. To test its capabilities, it is evaluated on a stratified sample of 100 newly found SonarQube issues from Mermaid.js, a large-scale open-source project, using a three-layer evaluation that combines the pipeline's own verdict, an independent LLM-as-judge and a post-remediation static analysis scan.

Together, these layers report a true remediation rate of 53%, rising to 57% once correctly assessed false positives are included. However, its changes were not always clean as it introduced new issues in 25% of the files it touched. The entire run used under half the memory of a single H100 NVL GPU, with each issue taking around seven to eight minutes to process.

Even though the system remains behind cloud frontier models on raw capability and remediation rate, recovering more than half of the flagged debt from a large-scale mainstream codebase with a comparatively small model is notable in itself. It shows that local open-weight remediation is a viable alternative on cost, data privacy and workflow fit, and can serve as an automated backlog-clearance tool, provided a human remains the final decision-maker on its output.

Keywords: technical debt remediation, open-weight language models, multi-agent systems, self-hosted inference, LLM-as-judge, static analysis, privacy

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background and Motivation

1.1.1 The Cost of Technical Debt in Practice

Imagine a scenario where a development team, working on a growing multi-tenant based system, is suddenly notified of a bug where a user's information from tenant A is leaking over to tenant B. A bug that is mission-critical for a system used by thousands of businesses. Because of sub-optimal code structure, poor maintenance practices and almost no documentation on the inner workings of the system, it takes the development team the whole day to figure out and patch the bug. Scenarios like these are common in modern software development, where teams under pressure face the choice of *doing it right* versus *doing it now*. Oftentimes, the choice is the latter [1].

These trade-offs, over time, manifest as what is known as **technical debt**. Empirical studies estimate that developers lose around 23% of their working time dealing with the consequences of technical debt, such as hard-to-understand code, fragile tests and workarounds for legacy design decisions [2], [3]. For organizations, this translates into substantial productivity losses and increased maintenance costs, making technical debt remediation a high-priority concern across the industry.

1.1.2 Agentic AI as a Solution

Recent advancements in artificial intelligence, particularly LLMs (Large Language Models) in an agentic workflow, have demonstrated promising capabilities in a wide variety of tasks including automated code repair and technical debt remediation [4]. However, frontier LLMs like GPT, Claude, Gemini etc. present significant limitations for widespread adoption. First, sending proprietary source code to third-party cloud providers raises serious concerns around intellectual property, especially for organizations in regulated sectors such as finance, healthcare and public administration. Second, cloud LLM APIs incur substantial operational costs and this becomes especially restrictive when running continuous or large-scale workflows. Finally, even though frontier LLMs are far more capable on most general tasks, their generalist nature makes them an overkill for most task-specific workflows including software maintenance.

These constraints have motivated a shift towards solutions with locally deployable language models within the organization's infrastructure. Critically, local deployment means that proprietary code and sensitive documentation never leave the organization's infrastructure. Among these, recent research by Belcak et al. from NVIDIA [5] suggests that SLMs (Small Language Models), compact models with around 10 billion parameters, can achieve comparable performance to their generalist LLM counterparts. Their compact size makes them especially economical and effective enough to be used in most agentic AI systems. This is especially true as a deep dive into these systems reveals that the atomic tasks are fundamentally simple, close-ended and repetitive [6].

1.2 Problem Statement

While technical debt remains a significant challenge in modern software development, and LLMs have shown promise in automating its remediation, a critical gap remains between compliance with third-party privacy policy and the practical constraints faced by most organizations. Specifically, the requirement for proprietary codebases to stay within the organization while still maintaining comparable quality to cloud-based frontier LLMs. This gap has motivated interest in locally deployable models, yet there is limited empirical evidence on their efficacy in software maintenance workflows especially for code-level and documentation-level technical debt. With SLMs showing promise, surely, other more capable locally deployable models can achieve better results.

1.3 Research Objective

This research aims to advance the field of AI-assisted sustainable software engineering by investigating whether locally deployable open-weight language models, organized in a multi-agent architecture, can effectively automate technical debt remediation with comparable performance to general-purpose Large Language Models. In doing so, the work will contribute empirical evidence on the accuracy-efficiency trade-offs inherent in using open-weight models for agentic workflow in software maintenance contexts.

In addition to that, this research also seeks to develop and showcase a privacy-focused, cost-effective and locally deployable solution to automated technical debt remediation within a resource-constrained environment. This will serve as a proof-of-concept and a much-needed alternative for organizations, particularly startups and open-source projects looking to boost their productivity with limited resources.

1.4 Research Questions

In order to effectively accomplish the objective of this research, this thesis attempts to answer the following research questions:

RQ1 How can a multi-agent system built on open-weight models be designed together with existing tools and workflows to remediate technical debt?

RQ2 How effectively can open-weight models remediate technical debt in an agentic workflow, especially in terms of remediation accuracy and code quality improvement?

RQ3 What are the computational resource requirements and operational costs of open-weight model based technical debt remediation?

RQ4 How feasible is the integration and deployment of local open-weight model based technical debt remediation in real-world development workflows compared to a generalist cloud-based LLM?

1.5 Scope of the Work

This thesis is a proof-of-concept study of automated technical debt remediation using locally deployable open-weight language models. To keep the work feasible within its time constraints, its scope is bounded in several ways. First, the prototype takes the issues reported by a static analysis tool and focuses on remediating code-level debt along with documenting the affected code. Detection is therefore left entirely to a static code analyzer of which many exist. However, for this thesis, the prototype uses only one, SonarQube [7]. The system thus consumes SonarQube's output rather than identifying the debt itself and targets the specific rule violations it flags rather than higher-level architectural or design debt.

Second, the study is limited to open-weight models. More specifically, it uses only those that fit the hardware of the University of Turku Department of Computing's AI inference server. Within that limit, a single model was selected from public benchmark leaderboards, filtered to the parameter range the hardware could serve and used for all agents. This single-model choice was made both to fit the available research time and to keep differences between models from affecting the results. The model was used as-is, without any fine-tuning.

Third, the empirical evaluation is limited to a single open-source project written mostly in a single language. To step up from previous work, however, the project is deliberately chosen to be large, with a correspondingly large user base and embedded in major developer tooling and platforms across the industry. This also means that the project will produce a large number of issues to resolve within the time limit. As such, a sampled subset of the issues is explored within this thesis. Thus, generalization to other languages, project styles, model sizes or issue volumes is left to future work.

Finally, the system is developed and evaluated as a research prototype rather than a production-hardened tool, so questions of large-scale deployment, security and long-term maintenance fall outside the scope of this thesis.

1.6 Contributions

This thesis makes the following contributions:

- **A privacy-preserving multi-agent remediation system.** A working pipeline that remediates SonarQube-identified technical debt entirely on locally served open-weight models. It integrates existing developer tools such as SonarQube, Git, GitHub and CI (Continuous Integration) and delivers its output as a conventional, reviewable pull request.

- **Design knowledge for building around local open-weight models.** A set of practical findings on where to delegate work to an open-weight model versus where to rely on deterministic programmable logic and tooling, including why full function replacement was chosen over unified diffs and how AST-based extraction and byte-offset splicing keep the model out of positional reasoning.

- **A three-layer evaluation methodology.** A method for measuring true remediation that combines the pipeline's own verdicts from the language model backed validation agent, an independent LLM-as-judge with full codebase access and a post-remediation static analysis scan, consolidated into a three-way agreement metric that captures genuine, developer-grounded correctness rather than the mere satisfaction of a static code analyzer's rules.

- **Empirical evidence on the accuracy, cost and feasibility of remediation with local open-weight models.** Quantitative results on how effectively an open-weight model remediates code-level debt, the computational and operational cost of doing so on a single GPU and the conditions under which the approach is a credible alternative to cloud frontier LLMs.

1.7 Thesis Structure

The thesis is organized into eight chapters.

Chapter 1 introduces technical debt in software development, current implementations of agentic AI and why privacy-focused, locally deployable agentic solutions are needed. It presents the problem statement, research objective, questions, scope, contributions and thesis structure.

Chapter 2 reviews related work on AI for technical debt management, tracing the

evolution from early detection and classification approaches to recent LLM-driven and multi-agent remediation. It concludes by identifying the research gap.

Chapter 3 presents the research methodology. It outlines the experimental setup, including the criteria for model and codebase selection and the evaluation metrics.

Chapter 4 presents the system architecture and describes how the multi-agent system is designed, including the design rationale, agent workflows, development tooling integration and the privacy-focused deployment model.

Chapter 5 presents the implementation, including the technologies used, the practical system realization and the deployment setup.

Chapter 6 reports the empirical results, including quantitative and qualitative findings from the three-layer evaluation and the computational resource cost.

Chapter 7 discusses the results, interprets them with respect to the research questions, examines limitations and threats to validity and outlines directions for future work.

Chapter 8 concludes the thesis by revisiting the motivation and the four research questions, summarizing the contributions and reflecting on the broader significance of local open-weight models in code remediation.

1.8 Declaration of the Use of AI

This thesis is an original work and the author is solely responsible for its intellectual content, analysis and conclusions. The author acknowledges that generative AI tools were used in a supporting capacity throughout the work. During the preliminary

stages, they aided topic exploration and ideation. During the thesis work itself, they were used for gathering and summarizing research literature, for improving textual clarity and coherence and as a coding assistant for constructing the prototype.

2 Related Work

2.1 AI in Technical Debt Management

Modern software engineering is undergoing a significant change with the emergence of AI and LLMs. Tools such as Cursor [8], Lovable [9] and Claude Code [10] have streamlined day-to-day development, allowing developers to focus more on higher-level design, bug fixing and optimization [4]. This shift is also reshaping the relationship between AI and technical debt management, particularly in the context of automated remediation. This chapter synthesizes recent work on AI applications in technical debt, identifies the current state of practice and positions this thesis within the existing research landscape.

2.2 Early works on AI-Driven Technical Debt Management

Historically, research in AI-assisted technical debt management has focused predominantly on detection and classification rather than automated remediation. A comprehensive study by Pandi et al. reviewed 15 key studies on artificial intelligence for technical debt management [11]. Their findings show that most AI applications target code-smell prediction, SATD (self-admitted technical debt) identification and debt prioritization rather than automated repair. These works established that AI

can analyze static code using metrics and textual hints to identify and rank technical debt. However, they generally stop at recommendations rather than resolution.

Tools such as SonarQube [7] and Snyk [12] follow a similar pattern. They provide extensive rule sets and metrics for identifying code smells, bugs and security vulnerabilities, but their output still requires manual intervention for remediation. This detection-to-remediation gap motivated later research into using modern neural models to recommend or automate technical debt removal strategies.

Prior to the widespread adoption of LLMs, Zampetti et al. introduced deep learning for technical debt management tasks beyond simple detection [11], [13]. Their work addressed SATD removal by predicting one of six removal strategies rather than only identifying the debt. To do this, the researchers proposed SARDELE (SAtd Removal using DEep LEarning), a hybrid neural architecture with a CNN (Convolutional Neural Network) for processing embeddings from SATD comments and an RNN (Recurrent Neural Network) for processing embeddings from the corresponding source code. The model achieved an average precision of about 55%, a recall of about 57% and an AUC of 0.73 across strategy types. This showed that SATD removal follows recurring, learnable patterns and that more advanced remediation methods are feasible.

2.3 Emergence of LLM-Driven Technical Debt Resolution

Sheikhaei et al. provide a large-scale empirical evaluation of proprietary and open-weight LLMs on SATD repair [14]. Using newly curated datasets of 58,722 items from Python repositories and 97,347 from Java repositories, the authors established new baseline metrics, namely BLEU-diff, CrystalBLEU-diff and LEMOD (Line-Level Exact Match on Diff), for assessing remediation quality across models. Their

results show that open-weight models can repair SATD, but performance remains modest. For example, Gemma-2-9B achieved 10.1% Python and 8.1% Java resolution on the EM (Exact Match) metric, while Llama-3.1-70B achieved the best LEMOD score among the open-weight models. GPT-4o-mini-2024-07-18 was the only proprietary model in the comparison and performed competitively, though at higher API cost.

A key limitation of this work is the evaluation criterion. Because SATD can often be removed in more than one valid way, exact match against a human-written fix is an imperfect proxy for usefulness. Even so, the study confirms that LLMs have a foundational capability for SATD repair and highlights the cost-performance trade-off inherent in automated code repair.

Khalil presents a more comprehensive case study of an LLM-driven agentic workflow for code-debt remediation [4]. Combining reports from static-analysis tools such as SonarQube and Snyk, the author fed them into Claude Sonnet 4 within an agentic framework integrated into Cursor. The workflow demonstrates that LLMs can achieve approximately 90% resolution of code-level technical debt such as code smells, bugs and security vulnerabilities when applied to a real-world TypeScript project. However, the study evaluates only a single proprietary project with a single proprietary model, limiting the generalizability of the results across languages, project scales and domain contexts.

Tornhill et al. propose ACE (Augmented Code Engineering) [15], a framework designed to address technical debt introduced by AI coding assistants that often produce code of unknown quality. Standard LLMs typically achieve only 37% precision in functional refactoring, largely because their generative nature and hallucination tendencies work against reliable repair. ACE uses contextual model selection together with a multi-stage validation layer that incorporates syntactic, code-health and semantic checks. By automatically discarding invalid refactoring attempts, ACE

increases refactoring precision from 37% to 98% while maintaining a recall of 52%. This makes it a strong example of automated guardrails for technical debt remediation.

Parallel to technical debt remediation, multi-agent AI architectures have been applied to other software-quality tasks. Bakane introduces Noesiz, a specialized multi-agent system for security vulnerability triage [16]. The system uses a core of specialized sub-agents, namely a False Positive Detection Agent, a Risk Scoring Agent, a Risk Explainer Agent and a Remediation Agent. Each is backed by Gemini 2.5 Flash. The agents are arranged in sequence and share a single context, so each agent's output informs the next. Together they reduced false positives in security findings by 35.3%. While this result concerns vulnerability triage rather than technical debt remediation directly, it suggests that decomposing a complex task into coordinated, specialized sub-agents can improve outcomes over a monolithic agent. However, Bakane's system also depends on a cloud-based proprietary LLM and therefore inherits the same API-cost and data-privacy concerns.

2.4 Research Gap

The studies discussed in the previous sections show a steady move toward AI-driven software maintenance. However, few have considered open-weight models for systematic technical debt remediation.

Sheikhaei et al. [14] compared several open-weight and proprietary models on SATD repair, but their evaluation focused on how closely a model reproduced the exact developer-written fix. This failed to capture the broader set of remediations that were correct but did not reflect the ground truth exactly. Khalil and Tornhill et al. [4], [15] moved closer to full remediation pipelines by combining static-analysis tools such as SonarQube, Snyk and CodeScene [17] with LLM-based agents. However, their systems relied on a single proprietary cloud model, which introduces

recurring API costs and privacy concerns. These concerns are precisely why open-source alternatives have been on the rise. OpenCode [18], unlike Claude Code, is a model-agnostic open-source coding agent by design that allows any language model to be plugged in. Yet these tools are purpose-built to be generalist, interactive coding partners rather than a specialized system for automated remediation.

Multi-agent design has also shown promise in adjacent software-quality tasks. Bakane's Noesiz system uses specialized sub-agents for security vulnerability triage and filters out roughly a third of the false positives [16]. For a domain as complex and sensitive as application security, this shows potential. It suggests that a multi-agent approach could be just as effective for technical debt remediation, but this has not yet been explored with a locally served, open-weight model.

Thus, this thesis positions itself to address the following gaps no prior work has combined:

- Open-weight language models for privacy-first local deployment.
- Multi-agent architecture.
- Integration with industry-standard static code analyzers.
- Addressing code-level technical debt resolution.

3 Methodology

3.1 Research Approach

This thesis employs Design Science Research (DSR) as its primary methodology. DSR is a well-established paradigm in information systems and software engineering research that focuses on the creation and evaluation of innovative artifacts that aims to solve identified practical problems [19]. These artifacts, for example models, methods or systems, aren't merely just the only output, but the knowledge generated about why and how the artifact solves the problem, under what conditions it is effective and what trade-offs it entails also contribute to DSR.

This methodology is appropriate here because the primary output is a working system whose evaluation is only meaningful once the artifact exists and because the research questions span both the design of the artifact (RQ1) and its empirical performance (RQ2, RQ3, RQ4). The research proceeds through three iterative cycles: the Relevance Cycle, which identifies the problem and defines requirements (Chapters 1 and 2), the Design Cycle, which constructs and refines the artifact (Chapters 4 and 5) and the Rigor Cycle, which actualizes the design in theory and contributes empirical findings back to the knowledge base (Chapters 6 and 7).

3.2 Research Design

Following the DSR paradigm, there are four phases to the research design which is further discussed in the following sub-sections. Briefly, these phases are as follows:

Phase 1 : Compiling structured JSON data of technical debt identified from a static code analyzer. Additionally, select model based on relevant metrics used in multiple publicly available language model leaderboard.

Phase 2 : Constructing the workflow pipeline.

Phase 3 : Apply remediation and aggregate results.

Phase 4 : Evaluation of remediation outcome using both LLM and human as judges.

3.2.1 Codebase Selection

The technical debt dataset compiled by [20] was initially considered as a source of subject projects but was ultimately excluded for two main reasons. First, many of the included projects far exceeded the scale practical for the experiment in terms of issue volume and repository size. Second, the dataset was produced using SonarQube 7.5 [7], a version that predates significant changes to SonarQube's rule set, severity model and issue taxonomy of modern versions. Using the original dataset would have introduced a mismatch between originally identified issues and those identified after remediation using SonarQube v26.5.0.122743 Community Edition, the version used throughout this experiment.

Instead, the subject repository was selected from the SEART GitHub dataset [21], a curated collection of open-source projects used in software engineering research. The following selection criteria were defined prior to querying:

1. The project needed to be TypeScript-dominant, aligning with prior work [4].

2. Have between 5,000 and 100,000 GitHub stars, ensuring widespread community adoption and real-world relevance while excluding the very largest projects that would be impractical to process.

3. Have between 7,000 and 100,000 lines of TypeScript code, aligning with the project size of prior work [4] while ensuring the codebase is complex and production-ready without exceeding computational feasibility for running the pipeline on it.

4. Not be a fork, so that only original codebases are considered.

5. Have a wiki, as a signal of active documentation and project maturity.

Querying the SEART dataset with these criteria returned 415 candidate projects, of which the ten most-starred are shown in Table 3.1.

Project	**Stars**	**TypeScript LOC**
mermaid-js/mermaid	88,438	70,891
thedotmack/claude-mem	80,646	86,271
nestjs/nest	75,662	90,147
abi/screenshot-to-code	72,780	11,509
eugeny/tabby	71,504	23,024
apache/echarts	66,493	99,622
pmndrs/zustand	58,171	7,687
marktext/marktext	56,929	66,131
Lum1104/Understand-Anything	51,984	30,620
chenglou/pretext	47,780	20,028

Table 3.1: The ten most-starred TypeScript projects from the SEART query, ranked by star count.

From these, the top result by star count was selected. This yielded **Mermaid-js** [22], a widely adopted open-source diagramming and charting tool that renders diagrams from Markdown-like text. Written primarily in TypeScript with JavaScript as a secondary language, at the time of this study the project comprised roughly 92,000 lines of TypeScript and 35,000 of JavaScript, about 127,000 lines of code in

total. This exceeds the 100,000 line upper bound, but the excess comes entirely from the JavaScript component. The TypeScript portion alone stays within range and is the primary evaluation target. Mermaid-js also integrates deeply with industry-standard tools [23] such as GitHub, GitLab and Notion and the repository had over 88,000 stars, 9,000 forks and 1,400 open issues. Taken together, these characteristics make Mermaid-js a production-grade project and a credible, representative subject for automated technical debt remediation research.

3.2.2 Issue Sampling

A full SonarQube scan of Mermaid-js identified 3,642 issues across 493 files, comprising 2,901 code smells (80%), 520 bugs (14%), and 221 vulnerabilities (6%), with severity breakdown of 1 BLOCKER, 433 CRITICAL, 1,522 MAJOR, 1,534 MINOR and 152 INFO.

The following is the JSON structure extracted and fed into the workflow:

```
{
  "project_id": "string",
  "git_link": "string",
  "commit_hash": "string",
  "measures": {
    "minor_violations": number,
    "reliability_rating": number,
    "code_smells": number,
    "security_rating": number,
    "sqale_debt_ratio": number,
    "sqale_index": number,
    "critical_violations": number,
    "sqale_rating": number,
```

```
    "blocker_violations": number,
    "info_violations": number,
    "bugs": number,
    "major_violations": number,
    "cognitive_complexity": number,
    "vulnerabilities": number,
    "line_coverage": number,
    "cyclomatic_complexity": number,
    "complexity": number
  },
  "issues": [
    {
      "id": "string",
      "rule": "string",
      "type": "string",
      "severity": "string",
      "component": "string",
      "action_message": "string",
      "start_line": number,
      "end_line": number,
      "effort_minutes": number,
      "description": "string"
    }
  ]
}
```


Given the research scope and computational constraints, a sampling strategy was necessary to keep experimentation feasible while still enabling meaningful conclusions. Thus, a stratified random sample of 100 issues was drawn from this filtered

pool. Prior to sampling, issues were first filtered to retain only those residing in production `.ts` or `.js` files, which meant any file whose path contained `spec`, `test`, `__tests__`, or `cypress` was excluded to prevent test artifacts from influencing remediation evaluation. This reduced the pool to 1,460 issues across 264 production files.

Stratification proceeded in two stages. First, issues were separated by their three issue types (code smells, bugs and vulnerabilities) with a minimum allocation enforced across all of them so they are well represented in the evaluation corpus. Second, within each type, remaining slots after the minimum allocation were proportionally set across the severity levels (BLOCKER, CRITICAL, MAJOR, MINOR and INFO) reflecting the natural severity distribution of the filtered pool. Next, within each severity stratum, issues were shuffled using a fixed random seed to ensure reproducibility, then sampled sequentially subject to a per-file cap of five issues. This limit prevents any single heavily-affected file from dominating the sample. Ultimately, this procedure was necessary because purely proportional sampling would have under-represented bugs and vulnerabilities given their small share of the population.

3.3 Model Selection

For AI model inference, the University of Turku Department of Computing's shared AI inference server `compai-01.utu.fi` is used. The server specifications are listed in Table 3.2.

Candidate models were evaluated by cross-referencing two complementary benchmark platforms. Arena [24] is a community-driven human preference leaderboard that ranks models based on blind pairwise comparisons across a range of tasks. LLM-Stats [25] aggregates and standardizes published benchmark scores across different models including open ones, enabling direct cross-model comparison on academic

Component	Hardware	Notes
GPU 1	NVIDIA H100 NVL	95,830 MiB (≈94 GB) VRAM
GPU 2	NVIDIA H100 NVL	95,830 MiB (≈94 GB) VRAM
CPU 1	Intel Xeon Gold 6526Y	
CPU 2	Intel Xeon Gold 6526Y	
RAM	512 GB	16 × 32 GB DDR5

Table 3.2: Specifications of the `compai-01.utu.fi` inference server.

benchmarks. The following filters and metrics were used:

Arena Filtered on the *WebDev (Overall)* track, which measures agentic AI performance on coding tasks involving multi-step reasoning and tool-use via human preference votes.

LLM-Stats Filtered *Parameters: 0-75B* and *Open* license and sorted descending by **SWE-bench Verified**.

SWE-bench Verified as a filter was especially chosen since it evaluates model's capabilities on solving a set of human-verified real-world GitHub issues. Additionally, the combined VRAM of both GPUs (187 GB) theoretically permits models up to approximately 70-75B parameters at full precision, accounting for KV cache and runtime overhead and thus, a parameter limit of 0-75B was set.

Arena doesn't contain the same filtering capabilities and having less number of models evaluated, resulted in some conflicting rankings, as the analysis below shows.

The comparison across both benchmarks, as recorded at the time of data extraction from these, reveals an important discrepancy. **GLM-4.7-Flash** ranks third on SWE-bench Verified (59.2%) yet scores higher than Qwen3.5-27B on Arena (1354 vs. 1347). This inversion reflects a fundamental difference in what each benchmark measures. Arena's metrics reward multi-step agentic tool use and human preference, while SWE-bench Verified rewards direct issue-level code repair against real test suites. For the remediation task in this thesis, SWE-bench Verified is the more appropriate signal.

Rank	Model	Params (Total / Active)	SWE-bench Ver.	Code Arena	Architecture
1	Qwen3.5-27B	27B / 27B	**72.4%**	1347	Dense
2	Qwen3.5-35B-A3B	35B / 3B	69.2%	1247	MoE
3	GLM-4.7-Flash	30B / 3B	59.2%	1354	MoE
4	Devstral Small 1.1	24B / 24B	53.6%	—	Dense
5	Nemotron 3 Nano 30B-A3B	31.6B / 3.2B	38.8%	—	MoE (Mamba)

Table 3.3: Top five open models, ranked by SWE-bench Verified score. Arena scores are from the WebDev (Overall) track. Dashes indicate no published score.

Qwen3.5-27B is selected as the inference model for all agents in the pipeline. It leads by 13.2 percentage points on SWE-bench Verified over the next comparable model (GLM-4.7-Flash) and achieves the highest coding score (23.9) among all single-GPU-compatible candidates [25]. Its dense architecture ensures all 27B parameters are active on every forward pass. The model was served through Ollama at Q4_K_M 4-bit quantization, a widely used scheme commonly recommended as a balanced default, trading only a small loss in quality for a large reduction in memory [26]. This brought the 27B weights to roughly 17 GB.

All pipeline agents use the same model rather than a heterogeneous assignment. This eliminates inter-model variability as a confound, ensuring performance differences can be attributed to prompting strategy and task structure rather than model capability. Using a smaller model for lightweight tasks like summarization and documentation is a natural direction for future work (see Section 7).

3.4 Evaluation Framework

To address RQ2 (effectiveness of the open-weight multi-agent system) and RQ3 (computational and operational cost), the artifact is evaluated through three complementary layers of evidence. The first is the pipeline's own checkpoint self-report built from the verdicts of the Validation agent. The second is an independent LLM-as-judge that reviews each change with full project context. The third is a post-remediation SonarQube scan that measures rule satisfaction rather than semantic correctness. A fix is counted as genuinely remediated only where all three agree and manual inspection is used to understand the cases where they do not. The judge layer is described first, followed by the metrics drawn from all three.

3.4.1 LLM-as-Judge

Rule-based metrics and the pipeline's own checks cannot capture the broader picture of code quality and usability. A remediation may satisfy the rule pattern syntactically while being semantically incorrect. The model can remove functionality or over-simplify logic in a way that alters behaviour. The pipeline's own Validation agent also shares the same model biases as the Remediation agent and cannot serve as an independent quality signal. To assess quality beyond the tool verdict and the pipeline's internal checks, an LLM-as-judge approach is applied.

LLM-as-judge is an evaluation methodology in which a capable language model acts as an automated reviewer, scoring or classifying outputs produced by another model [27]. Given a structured prompt containing the artifact to be assessed and a well-defined rubric, the judge model returns a structured verdict. This approach has been shown to correlate well with human expert judgement for code quality tasks when the judge model is sufficiently capable and the rubric is specific [27]. It simplifies the evaluation of outputs that are too large in numbers to review entirely by a single human reviewer, all while preserving a semantic understanding that

rule-based metrics cannot provide.

Thus, for this experimentation, Claude Code with Claude Sonnet 4.6 serves as the semantic correctness judge, applied through the VS Code extension and Claude Pro subscription.

The judge operates at the pull request level, comparing the diff between the fix branch and the original base branch. For each applied fix, it assesses whether the change correctly addresses the stated SonarQube violation, whether it preserves the original function's observable behaviour and whether it introduces any new problems. It returns a structured verdict for each fix, one of *correct*, *partially correct*, *incorrect* or *unsafe*, together with a brief written justification. This is the per-issue semantic correctness signal.

The judge is also used to check the issues the pipeline declined as false positives. For each of these, it is asked independently whether the flagged issue genuinely requires a code change or was correctly left alone. This verifies the pipeline's triage rather than its fixes, and a correct decision to leave code untouched is counted as a correct outcome.

3.4.2 Evaluation Metrics

The metrics draw on three data sources. These are the per-issue checkpoint log the pipeline writes during a run, the SonarQube scans taken before and after it and the judge verdicts described above. Following [4], quality is assessed largely through the delta in SonarQube measures. The metrics cover remediation effectiveness, code quality and computational cost.

Remediation effectiveness:

Issue Resolution Rate Proportion of sampled issues the pipeline itself marked as resolved, taken from its checkpoint self-report. Reported in aggregate and broken down by issue type (Bug, Vulnerability, Code Smell) and by severity

(BLOCKER, CRITICAL, MAJOR, MINOR, INFO) to show whether the rate varies with issue characteristics.

Scan-Confirmed Resolution Rate Proportion of the sample that the post-remediation SonarQube scan confirms closed, measuring the rule satisfaction.

True Remediation Rate Proportion of the sample confirmed by all three layers, the Validation agent, the SonarQube post-scan and the LLM judge. This is the strictest measure and the primary answer to RQ2.

Correct Handling Rate True remediation extended with the false positives the pipeline correctly declined, each verified by the judge. It captures how often the pipeline did the right thing, whether by fixing an issue or by correctly leaving it alone.

Triage Precision Of the issues the pipeline declined as false positives, the proportion the judge independently confirms genuinely needed no code change. Reported separately from remediation, as triage and repair are distinct capabilities.

Code quality and regression:

Regression Rate Proportion of remediated files that introduce net-new SonarQube issues not present in the pre-remediation baseline scan. A fix that resolves its target while introducing new violations is treated as a low-quality resolution regardless of its status.

Complexity Delta Change in SonarQube's cognitive and cyclomatic complexity after remediation. A technically valid fix that substantially increases the cognitive burden of the code is not an improvement in code health.

Quality and Debt Delta Change in SonarQube violation counts (code smells, bugs, vulnerabilities and the Critical, Major and Minor severities) and in estimated technical debt, measured before and after the run both project-wide and scoped to the sample. The debt figure is SonarQube's own estimated remediation effort in minutes and does not reflect real developer effort.

Computational and operational cost:

Remediation Time Time taken to process each issue, reported as the mean per issue and broken down by outcome and by the language-model agents (Summarizer, Remediation, Validation and Documentation) to show where processing time concentrates.

Hardware Utilization GPU utilization, VRAM footprint and CPU utilization recorded on the inference server across the run, characterizing the operational cost of self-hosting the model.

4 System Architecture

4.1 Design Rationale

The system architecture to be introduced in this chapter is shaped directly by the limitations identified in the preceding review of related work. This section makes that connection explicit, explaining how each major design decision follows from a specific gap in the existing literature.

Industry standard static code analyzers are already sufficient at detecting code-level technical debt and assessing code health. They provide structured reports that aggregate technical debt findings with rule identifiers, severity levels, file locations, effort estimates and many more. This structured output serves as a well-defined entry point to the system. Thus, following the same route as Khalil's work, the proposed system should utilize existing tools to only focus on remediating, instead of constructing an independent detection system as an extra layer of complexity.

Prior systems treated cloud API access as an integral part of their architecture. This thesis inverts that requirement with local deployment being the primary constraint. Open-weight models are particularly well-suited to this requirement. Unlike closed API models, their weights are public, their behavior is reproducible across runs with fixed random seeds and they can be served entirely offline through local inference. This ensures no source code, intermediate artifact or generated output is transmitted to an external service that is not approved of while fulfilling the

requirement towards privacy.

Another key architectural consideration was going with a multi-agent system. A comprehensive study by Liu et al. (2024) of over 124 LLM-based agentic systems for different software engineering tasks found that single general purpose agents are generally inadequate for complex end-to-end software maintenance tasks [28]. With a multi-agent architecture, a key benefit is that each agent solves an atomic responsibility and maintains its own focused context window. For code-level technical debt remediation, the ability to understand what related dependent code blocks do matters more than the entire codebase and providing only the necessary information to these specialized agents prevents burdening a single model context. To further keep context small, the generated output can be code patches instead of a complete code block rewrite. Additionally, for tasks that have deterministic, computable output, a rule-based logic instead of language models can minimize system resource requirement and reduce inference timing while reserving language model inference for tasks that genuinely require their capabilities.

Taking inspiration from ACE [15], the proposed system needs to address the correctness of the proposed fix with a validation layer. As observed in the previous chapter, there are multiple approaches to repaying technical debt and a generated patch that appears structurally plausible may still fail to resolve the flagged issue, introduce new violations or break existing behavior elsewhere in the codebase. The validation layer therefore needs to address multiple dimensions of correctness including syntactic validity of the patched file, behavioral preservation verified through the existing test suite and validity from static analyzers that the fix resolved the issue. Patches that fail any of these checks return a structured rejection reason that can be used as context for the next remediation attempt, making retries more informed rather than random repetitions of the same inference.

While ACE fixes issues in the IDE for immediate single or small batch to be

reviewed by the developer, the proposed solution needs to integrate into an existing CI/CD pipeline and remediate an accumulated technical debt backlog asynchronously while leaving human review in the loop. Thus, an ideal solution for the proposed system is to utilize VCS (Version Control System) to keep structured auditable trail to be reviewed, fixed or even rolled-back by the developer while showcasing a structured summary of all fixes attempted, including those requiring human intervention.

4.2 Overview of the Proposed Agentic Workflow

This section presents the system architecture of the proposed privacy-focused, agentic AI workflow for automated technical debt remediation. As depicted in Figure 4.1, the system is designed around a multi-agent based pipeline in which a structured report of technical debt findings extracted from a static analysis tool is fed through a set of specialized agents, processing them sequentially to produce working and well-documented code patches. Ideally, no source code, documentation, or intermediate artifact leaves the deployment environment of the organization.

The architecture follows the principle of agent atomicity meaning each agent has a single, well-defined responsibility and communicates with other agents exclusively through structured data contracts. This design with separation of concern in mind, enables each agent to be tested, fine-tuned or even replaced independently making ablation studies where individual agents can be disabled or swapped to measure their contribution to overall system performance a possibility.

The pipeline is organized into seven agents: an Orchestrator, a Context Agent, a Summarizer Agent, a Remediation Agent, a Validation Agent, a Documentation Agent, and a VCS Agent. The Orchestrator acts as a central coordinator delegating every task to its subsequent agents and keeping log of the remediation status. The remaining agents are mostly stateless workers that receive a well-defined input,

perform their designated task, and return a structured result either the next agent or back to the Orchestrator.

The system accepts a SonarQube JSON report as its sole external input. All subsequent agentic operations are performed locally using open-weight language models served via Ollama and rule-based tooling.

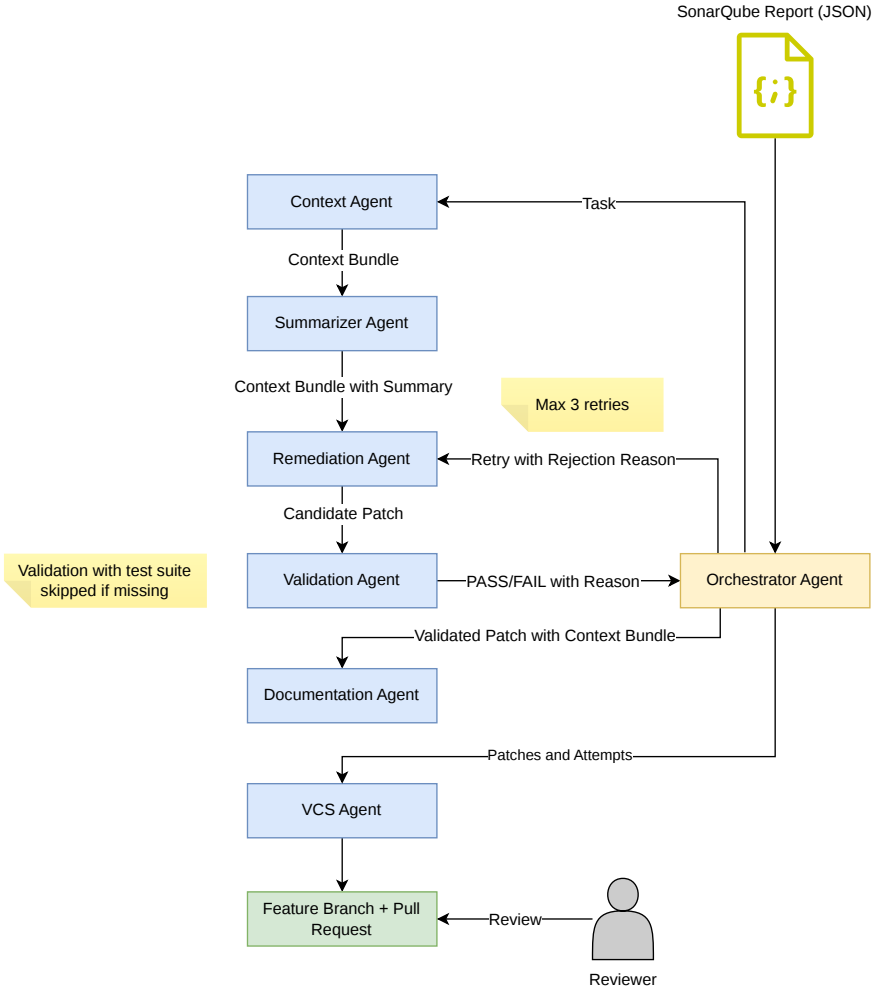

Figure 4.1: Proposed Multi-agent System Architecture

4.3 Agent Roles and Interactions

Orchestrator Agent The Orchestrator Agent is the central supervisor of the pipeline, implemented as a rule-based state machine with no language model inference.

It parses the SonarQube report at startup, constructs a prioritized task queue, and dispatches each task into the agent pipeline. Priority is computed from a weighted combination of cognitive complexity score, code churn from Git history, and test coverage of the flagged file. The Orchestrator also de-duplicates issues at initialization, grouping instances of the same rule pattern across multiple files as related tasks and reusing cached rule documentation across them.

The Orchestrator re-enters the pipeline only at decision points, specifically after the Validation Agent returns a result. If validation passes, it forwards the task to the Documentation Agent. If validation fails and retry attempts remain, it re-dispatches to the Remediation Agent with the structured rejection reason appended to the context bundle. If attempts are exhausted, it marks the task as FAILED. Once all tasks have reached a terminal state, the Orchestrator issues a single dispatch to the VCS Agent containing both the approved patches and the failed tasks with their rejection histories.

Context Agent The Context Agent assembles the context bundle passed to downstream agents. Given a task object, it extracts the full source of the flagged function or class, the file's import block, and a flat list of all function and class names already defined in the file. It also retrieves the appropriate test functions if applicable. The bundle contains everything deterministically retrievable from the local codebase that are relevant before asking the Summarizer Agent to describe what the function does.

Summarizer Agent The Summarizer Agent has a single responsibility: given the target function as source, produce a concise plain-English description of what the function does, what are it's parameters and what it returns. This summary serves as an explicit statement of the behavior that must be preserved if the Remediation Agent attempts structural refactoring. This summary is also

passed to the Documentation Agent as the base docstring after the fix is applied.

Remediation Agent The Remediation Agent is the primary inference component of the pipeline. It receives the bundled context and generates a unified diff patch resolving the flagged issue while preserving the described behavior. The output is constrained to a unified diff rather than a full file rewrite, keeping the output compact and unambiguous for downstream validation. The agent is instructed not to change public function signatures, not to introduce imports absent from the existing import block, and not to use names already present in the file when extracting helpers.

On retry attempts, the Orchestrator appends the structured rejection reason from the previous validation failure to the context bundle before re-dispatching. Each successive attempt therefore has strictly more diagnostic information than the previous one, making retries substantially different rather than a random repeated outcome of the same inference.

Validation Agent The Validation Agent verifies the correctness of a candidate patch through three checks. First, the patched file is parsed for syntactic correctness. Second, the test functions identified by the Context Agent are executed against the patched file to ensure that the core functionality after changes remain intact. Third, the patched file is re-analyzed by SonarQube. If at any stage, a failure occurs, the patch is rejected without running subsequent checks.

The Validation Agent returns a binary result and a structured reason to the Orchestrator, which makes all routing decisions. In the absence of the project's existing test suite, the second check simply doesn't run, a key limitation which is discussed further in Chapter 7.

Documentation Agent The Documentation Agent is invoked by the Orchestrator after a patch has been validated. It receives the approved patch, the original function source, the plain-English summary from the Summarizer Agent, and the list of any new function names introduced by the patch. It writes or updates the docstring of the refactored function using the summary as its base, generates docstrings for any newly extracted helper functions, and produces a concise changelog entry for the PR description.

VCS Agent The VCS Agent is the final component in the pipeline. It receives a single dispatch payload from the Orchestrator containing the list of approved patches with documentation and the list of failed tasks with rejection histories. It creates a feature branch, applies each approved patch as an atomic commit with a structured message, and opens a pull request against the main branch. The PR (Pull Request) description includes resolved and failed issues and a validation summary. The PR is never auto-merged and requires a deliberate action from the developer to verify the work before merging.

4.4 Integration with Development Tooling

The system integrates with three development tooling: static analysis, version control, and the local language model serving layer.

Static analysis integration is achieved through SonarQube's JSON report export and rule documentation API. The pipeline consumes the exported report as its sole input. It is also invoked by the Validation Agent for re-analysis of patched files, using the same project configuration and quality profile as the original scan. This ensures that the acceptance criterion for a fix is consistent with the project's established quality standards.

Version control integration is handled by the VCS Agent using a Git installa-

tion. The agent requires write access to the repository to create branches, commits and API access to the project's hosting platform (GitHub or GitLab) to open pull requests programmatically. All Git operations are performed locally or in remote hosting platform which in most cases is hosted by the organizations themselves.

The language model serving layer uses Ollama as the local inference runtime for all open-weight LM agents. Ollama manages model loading, GPU memory allocation, and request queuing.

5 Implementation

This chapter documents the technical realization of the multi-agent pipeline introduced in Chapter 4. The chapter goes into implementation details covering how agents share state, how each agent performs its own tasks including function code block identification and extraction, how each language model agent in the per-function loop operates and finally how completed patches are committed and a pull request opened. Implementation divergence from the proposed design is inevitable due to engineering and language model limitations and those are explained with rationale in place and collected in a summary table at the end of the chapter.

Figure 5.1 shows the realized system architecture. From extracting the SonarQube issue report, running remediation attempts through the multi-agent pipeline to opening a pull request against the target repository's main branch, the whole system is packaged as a Docker container intended to run as part of a GitHub Actions workflow. It is available in the author's GitHub[1].

5.1 Technology Stack

Table 5.1 lists the core technologies used and the role each plays.

While most of these are conventional choices, two of them merit explanation. First, as per the proposed system design, the Orchestrator makes most routing decisions. As such, a graph framework better fits the requirements than sequential

[1]https://github.com/Artorias17/Multi-Agent-Technical-Debt-Remediator

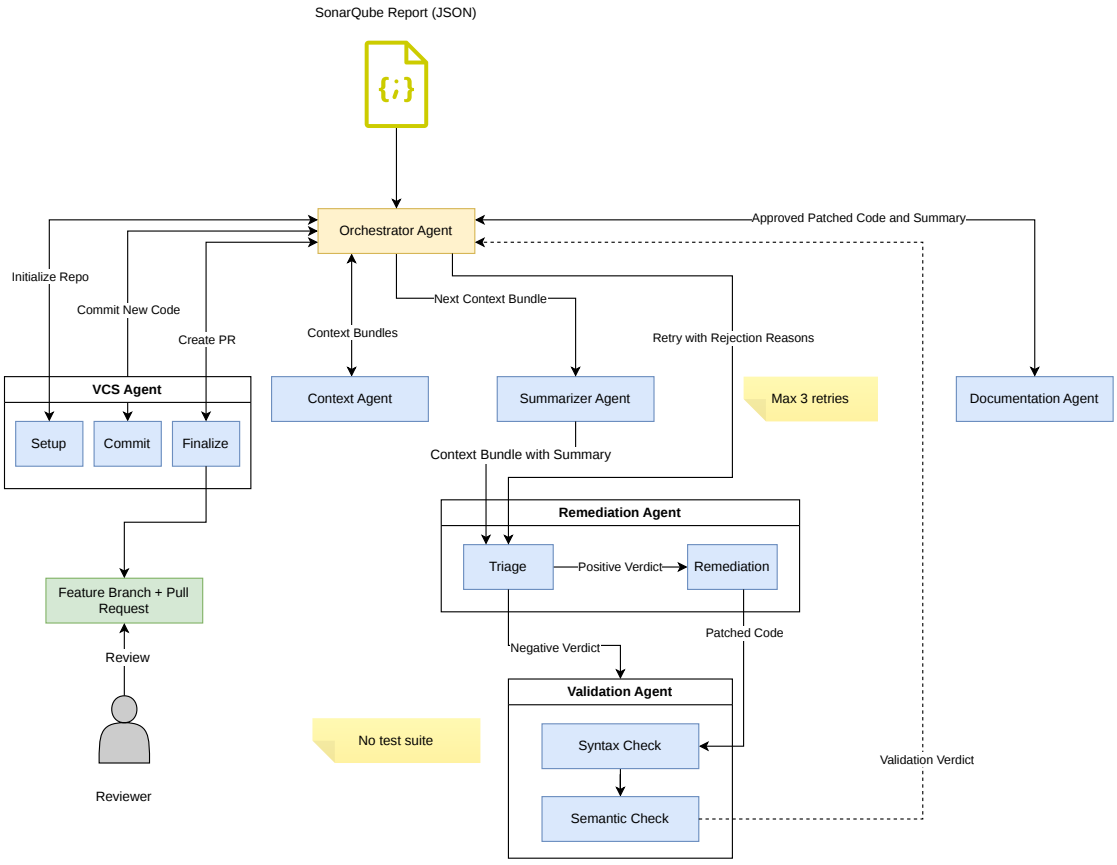

Figure 5.1: Final Multi-agent System Architecture

Component	**Technology**	**Role**
Language	Python 3.14	Implementation language
Pipeline framework	LangGraph	Typed StateGraph, conditional routing and shared state management
Model runtime	Ollama	Local model serving exposed via an OpenAI-compatible `/v1` endpoint
Model client	OpenAI Python SDK	Inference calls to Ollama's endpoint
Language model	Qwen3.5-27B	Backbone model for all agent inference
Code parsing	tree-sitter	Multi-language code parsing, function extraction, splicing and syntax validation
Version control	GitPython	Local branch and commit operations
PR integration	PyGitHub	Pull request creation via the GitHub API

Table 5.1: Implementation technology stack.

code. LangGraph [29] was chosen because its `StateGraph` abstraction separates each agent into a node and makes routing explicit. Agents communicate through a shared typed state rather than through direct calls, following the system design outlined in Chapter 4. Second, the OpenAI Python SDK serves as the inference client to Ollama's OpenAI-compatible endpoint. This makes the model backend substitutable simply through an environment variable.

5.2 SonarQube Report Extraction

A standalone preprocessing script, `fetch_report.py`, pulls a project's scan results from the SonarQube Web API beforehand and serialises them into a single JSON report that becomes the pipeline's sole external input. It collects the commit hash of the scanned revision so the exact code state can be checked out, the project-level debt and quality measures, all unresolved issues and the description of

every rule those issues reference. Each issue is then reduced to the fields the pipeline consumes, with its rule description embedded inline. Because of this, the pipeline does not need to consult with SonarQube during a run, which keeps the remediation stage self-contained and reproducible from the report file alone.

This is the first divergence from the proposed design. The architecture in Chapter 4 relied on SonarQube at runtime in two places. First, the Orchestrator was to query and cache the rule documentation API every time it starts working on a new issue. Second, the Validation Agent was to re-invoke SonarQube to confirm each fix against the project's quality profile. Fetching every rule description ahead of time and embedding it into the report addresses the first of these, so the pipeline makes no rule lookups against the analysis server while running, essentially speeding up the process. The Validation Agent's re-analysis is a separate runtime dependency and is discussed together with its replacement in Section 5.6.4. Both divergences are recorded in Table 5.4.

As presented in Chapter 3, to keep the run computationally feasible while preserving a representative spread of issue types and severities, the fetched report is not directly passed to the pipeline. Instead, a second preprocessing script, `stratify.py`, restricts the issues only to the actual source files and excludes issues from test and specification files. It then draws a sample of 100 issues, stratified by issue type and then by severity within each type. A max limitation of five issues per file is enforced so that no heavily affected file dominates. For reproducibility, this strategy went through a fixed random seed.

5.3 Pipeline State and Data Flow

The proposed design in Chapter 4 defined seven agents. The realized pipeline implements them as nine LangGraph nodes with the two extra arising from splitting the VCS Agent into separate setup, commit and finalize stages for reasons given in

Section 5.7. All nine share a single typed state object, `PipelineState`. Each node reads the fields it needs, performs its work and writes its results back before returning. Table 5.2 summarises what the state carries.

Group	**Fields**	**Purpose**
Input	`report`	The SonarQube report, the pipeline's sole external input
Work queue	`chunks`, `chunk_index`, `functions_to_process`, `function_index`	Position in the outer file loop and the inner function loop
Active function	`context`, `current_issues`, `summary`	The code, issues and behavioural contract the agents work from
Proposed fix	`replacement`, `helpers`, `patched_code`	Generated function source, any extracted helpers and the resulting file
Retry state	`attempt`, `rejection_history`	Attempt counter and the accumulated reasons earlier attempts were rejected
Outcomes	`last_event`, `approved / failed`	Routing signal from the last node and the cumulative resolved and failed records
Bookkeeping	`repo_path`, `branch_name`, `checkpoint_path`	Local clone, active feature branch and checkpoint file

Table 5.2: Fields of `PipelineState`, grouped by the role they play in a run.

Because of the shared state system, routing is designed to be event-driven and the Orchestrator is the only one that reads these signals, preserving agent atomicity throughout the system.

5.4 Orchestrator

Following the system design, the Orchestrator is implemented as a rule-based state machine with no language model backend. It first has the VCS agent initialise a local copy of the codebase and create a feature branch (Section 5.7), then drives the pipeline through two nested loops: an outer loop over file chunks and an inner loop

over the function groups within each chunk.

5.4.1 Chunk Building

Once the feature branch is ready, the Orchestrator groups the report's issues by file so that each file and the issues within it form one unit of work. Units are then ranked by the severity of the worst issue they contain and when they rank equally, the one carrying more issues goes first.

This is a simplification of the proposed design, which ranked work by a weighted combination of cognitive complexity, code churn and test coverage. Since those metrics require Git history and coverage data that the SonarQube report does not inherently carry on its own and construction for their support adds more complexity than it benefits in the actual remediation process, the priority is instead derived from the severity and issue counts already present in the report.

5.4.2 Function Processing Loop

For each file, the Orchestrator calls the Context Agent, which returns an ordered list of function groups, each pairing one function with the issues that fall inside it. The Orchestrator works through this list, passing every function to the Summarizer, Remediation and Validation agents in turn. Functions already recorded in the checkpoint are skipped, so an interrupted run can resume without repeating completed work.

Similar to the system design, the Orchestrator's handling of Validation outcomes determines the branching behaviour and dispatch of work to the next agent. Four cases arise:

Validation passes with a fix applied The Documentation agent is dispatched. On completion the Orchestrator records the function as resolved, adds the

signatures of the revised function and any new helpers to the file's name inventory.

Validation passes with no fix needed The Remediation agent's triage step judged the issue as a false positive. The function is recorded as unresolved and the Documentation agent is skipped.

Validation fails with retries remaining The rejection reason is appended to the rejection history field of the shared state and the Remediation agent runs again with new information.

Validation fails with retries exhausted All permitted attempts have failed. The function is recorded as unresolved and the Orchestrator moves on.

Once every function in a file has been processed, the file's changes are committed before the next file begins.

5.5 Context Agent

The Context Agent reads the target file from disk and assembles the bundle of information that all downstream language model agents consume. Its three responsibilities are function extraction, issue-to-function grouping and bundle assembly.

5.5.1 Function Extraction

Before a function can be extracted, the agent needs to know how to read the file. The language is inferred from the file extension, with TypeScript, JavaScript, Python, Java and C# currently supported, each registered with its own grammar. Supporting a further language therefore requires little change to the agent's logic.

The key engineering challenge is deciding which code block to pass downstream. A function is the natural choice, since it is syntactically complete, self-contained

and represents a single behaviour. It is also the code block most commonly tested in unit testing to confirm that behaviour matches its specification [30]. Smaller fragments would leave the model without the context needed to preserve behaviour, while whole files would exhaust the context window.

Function boundaries are identified using tree-sitter, a parser generator that builds a concrete syntax tree for a source file [31]. Parsing is necessary because files in a real codebase are not formatted uniformly. Signatures may span several lines, brace and indentation styles differ between contributors and projects, and declarations may sit inside classes, closures or export wrappers. Variations like these are difficult to interpret with line-based or regular-expression matching. Beyond handling this variation, two of its properties prove valuable further down the pipeline. Tree-sitter is error-tolerant, producing a usable tree even when a file does not parse cleanly, and every node carries the exact byte offsets of the text it spans, allowing a function to be located and later replaced without relying on line numbers.

Every grammar names its own node types for functions, listed in Table 5.3. To find the function an issue belongs to, the agent searches the parse tree for nodes of these types and keeps the smallest one containing the flagged line.

Language	**Function node types**
TypeScript, JavaScript	`function_declaration`, `method_definition`, `arrow_function`
Python	`function_definition`
Java, C#	`method_declaration`, `constructor_declaration`

Table 5.3: Tree-sitter node types searched when locating a function, by language.

For a normal function declaration this node is the whole function, so it forms the boundary directly. Arrow functions assigned to variables are harder, because the matched node covers only the right-hand side of the assignment. The agent therefore walks up to the enclosing declaration and, if that declaration is exported, one level further so that modifiers such as `export` and `default` fall inside the

range.

For issues that fall outside any detected function, the agent falls back to a short window of lines centred on the issue. These fallback extractions are handled by a separate path in the Remediation and Validation agents that skips helper extraction and substitutes by line range rather than byte offset.

5.5.2 Issue-to-Function Grouping and Processing Order

Once the boundaries are known, each issue is assigned to the function containing it. The resulting function groups are then processed from the bottom of the file upward. This ordering is deliberate as editing a file shifts the position of everything below the edit point, so working upward leaves the boundaries of the remaining functions untouched.

5.5.3 Context Bundle

The bundle follows the same design as in Chapter 4, carrying the file path and language, the full file source, the extracted function with its recorded boundaries and the file's import block. Two things differ.

The first is the name inventory, which holds more than the flat list of names originally specified. It maps each function in the file to its signature, which the Remediation agent consults so that any helper it introduces does not collide with an existing name or even the signature. The Orchestrator refreshes it after each documented function so that helpers added early in a file are visible to later functions in the same file.

The second is that test functions are not retrieved, as proposed in the design phase. While this limits the capabilities of the Validation agent, running them requires each project's dependencies to be installed and its test runner configured, neither of which the pipeline can derive from the report. Furthermore, a full test

run is also too slow and redundant to repeat for every individual function edit.

5.6 Language Model Agents

5.6.1 Shared Inference Design

All four agents with a language model backend route their requests through a single shared configurable API wrapper. Two additional parameters beside the model are also configured, which govern the token budget. First, `max_tokens` sets the token budget for the response. It is capped at 8,192 tokens, enough for a function rewrite with additional helpers and the reasoning tokens emitted beforehand. Second, `num_ctx`, which sets the context window length, controls the total tokens allocated to both the prompt and the response together. It is set to 32,768 tokens. This 3:1 ratio ensures that the context bundle with the target function leaves enough room for the language model to respond fully. As the latter is not a standard OpenAI parameter, it is passed to Ollama through the client's `extra_body` field [32]. Even though the model, Qwen3.5-27B, supports a native context window of 262,144 tokens [33] which is extensible even further, the configuration above gives a good balance of key-value cache memory and latency.

Each agent is driven by a zero-shot prompt that assigns it a role suited to its task along with the target function, its issues and the behavioural contract directly. Additionally, the prompts were zero-shot, meaning no worked examples were provided to the language model. Such role-based prompt designs are well-established practice for software engineering multi-agent systems [34] and are a recognised technique [35] shown to improve reasoning over plain zero-shot prompting [36]. This matters most for the triage, remediation and validation steps, where the model must reach a reasoned judgment and produce a working fixed code block rather than merely generate text. All of the prompts are available in Appendix A.

Agents are also instructed to respond only with valid JSON and to omit any prose or markdown fences, giving a consistent output format for a shared parsing function to decode. Because the raw response is not always clean, that function recovers the result defensively: it first strips any reasoning block, such as the `<think>...</think>` content the model emits before its answer, then attempts a direct parse and, failing that, searches the response for the first well-formed JSON object.

Temperatures are set per agent. The Summarizer, the triage step of the Remediation agent and the Validation agent all perform assessment rather than generation and run at 0.1. Patch generation runs at 0.2, allowing slightly more variation in how a fix is expressed, which reduces the chance of a retry reproducing the same failure.

5.6.2 Summarizer

The Summarizer for the most part is similar to the design proposed in Chapter 4. Given the extracted function, it returns a structured description of what the function does, its parameters and its return value. This description reaches the Remediation agent as the behavioural contract a fix must not violate, and the Documentation agent as the basis for the function's docstring. Because the contract is established before any fix is attempted and left unchanged across retries, every remediation attempt is judged against the same definition of correct behaviour.

5.6.3 Remediation

The Remediation agent is where divergence happens the most. First, in place of the single inference the design specified, it decomposes the remediation task into two sub-tasks as follows.

Triage The model is first asked to assess the issue and determine if it genuinely requires a code change. If not, the agent stops and reports the issue as needing

no fix, sparing the generation token budget for actual addressable issues. This step produces the false positive verdicts reported in Chapter 6.

Patch generation If a fix is needed, however, the model receives the function with the full context bundle including the function's behavioural summary and any accumulated rejection reasons to use as textual feedback that mirrors feedback-driven optimisation of agent prompts in multi-agent systems [34]. Other than that, the agent is asked to follow the same constraints as in the system design.

The second divergence is the shift to a full function replacement. While the proposed design specified a unified diff for compactness, in practice the model could not reliably produce correct diffs. Hunk headers carried wrong line numbers, context lines were hallucinated or omitted and off-by-one errors appeared even when the logical change was sound. The model understood what needed to change but could not encode it in a format demanding precise positional arithmetic. Full function replacement, adopted after iterative testing, removed the problem entirely. However, this came at a cost. A whole function, especially a larger one, uses far more of the response budget than a diff, resulting in the model occasionally returning an empty or truncated result.

5.6.4 Validation

The proposed design validated a patch by parsing the whole patched file, executing the file's tests and re-analysing it with SonarQube. The implementation changed some steps in this scheme. First, test execution is absent because the tests are never retrieved (Section 5.5). Second, SonarQube re-analysis is replaced by a language model-based semantic review. For a single function rewrite, re-scan of the whole project every time is a redundant endeavour.

Validation therefore runs in three stages that short-circuit on the first failure. The replacement is first spliced into the file in-memory within the boundaries of

extraction or substituted by line range for a fallback snippet. It is then parsed on its own to confirm it is syntactically valid, quickly identifying if any malformed output was produced when the model exhausts its token budget. Finally the replacement and its issues are returned to the model, which is asked whether each issue is resolved and whether the fix introduces new problems.

5.6.5 Documentation Agent

The Documentation agent follows the proposed design closely. Invoked after a patch passes validation, it receives the approved replacement source, the original function source, the Summarizer's behavioural description and the names of any helpers introduced by the patch. It generates or updates the docstring of the primary function using the Summarizer output as its base, generates docstrings for each helper and produces a conventional-commit message. The documented file is written back to disk, overwriting the version left by the Validation agent, and the commit message is carried in the shared state for the VCS commit stage to use when the file is committed.

5.7 VCS Agent

The VCS component is realized as three separate nodes rather than a single agent dispatched at the end of the run. Splitting it this way lets completed work be committed incrementally throughout the run and allows the checkpoint system to resume the pipeline if it stops abruptly.

The three nodes run at different points in the pipeline:

Setup Runs before any issue is processed. It opens the repository, cloning it at the commit recorded in the report if no local copy is supplied and creates a timestamped feature branch. When resuming, it instead checks out the

existing branch and commits any changes the previous run left behind.

Commit Runs after all functions in a file have been processed, staging the file and making a single commit whose message lists every issue resolved and incorporates the Documentation agent's message where one exists. Empty or unapproved files are skipped. Committing once per file rather than once per function keeps the history at a meaningful granularity, each commit being a self-consistent set of changes to one file.

Finalize Runs once every file is committed, pushing the branch and opening a pull request against the repository's default branch. Its body is built from the checkpoint file rather than in-memory state, so the summary reflects the full run history including work from an earlier session before a resume. The pull request is left open for a developer to review.

5.8 Checkpoint System

Throughout a run the pipeline appends each processed issue, with its outcome and per-agent timings, to a checkpoint file. On startup, if a checkpoint and a branch to resume are supplied, the Orchestrator skips any function whose issues are already recorded. This lets the remediation process be restarted after an interruption without duplicating work.

5.9 Summary of Divergences from Design

Table 5.4 summarises where the realized implementation differs from the proposed system architecture in Chapter 4.

Aspect	**Proposed Design**	**Implementation**
Prioritisation	Cognitive complexity, Git churn and test coverage	Maximum severity weight and issue count per file, sourced from the SonarQube report
Rule documentation	Fetched from SonarQube API per rule at runtime	Embedded in issue records by `fetch_report.py` before the run; no runtime SonarQube connectivity required
Test function retrieval	Retrieved by Context Agent and passed downstream	Not implemented
Remediation process	Single inference	Triage call, then a patch generation call
Remediation output	Unified diff	JSON: `replacement` (complete function source) and `helpers` array
Validation check 1	Syntax parse of patched file	Byte-offset splice applicability; syntax check follows as check 2
Validation check 2	Test execution	Syntax check of the replacement in isolation
Validation check 3	SonarQube re-analysis of patched file	Language model semantic review of replacement and helpers
VCS structure	Single agent dispatched at end of run	Three nodes: setup at start, commit per file, finalize at end
Commit granularity	One commit per approved patch	One commit per processed file

Table 5.4: Divergences between the proposed system architecture and the realized implementation.

6 Results and Analysis

This chapter presents and analyses the results of running the open-weight multi-agent pipeline on the 100 sampled SonarQube issues from Mermaid.js, as described in Chapter 3. The findings address two research questions: RQ2, how effectively the system remediates SonarQube-identified technical debt, and RQ3, the computational and operational cost of doing so.

To answer RQ2, three complementary layers of evidence are assembled in sequence, each corresponding to one section of this chapter. Section 6.1 reports the pipeline's own checkpoint outcomes, disaggregated by severity and issue type. This is the system's self-reported view, but the whole pipeline runs on a single model and configuration, so its Validation and Remediation agents share the same model biases, token budget and contextual scope. It therefore cannot serve as an independent signal of effectiveness. Section 6.2 addresses this with a post-hoc LLM-as-judge evaluation, in which Claude Sonnet 4.6 independently reviewed each issue with full access to the Mermaid.js codebase, judging whether fixes were semantically and behaviourally correct rather than merely plausible. It also verified the false positives the pipeline had flagged. Section 6.3 adds a third perspective through a post-remediation SonarQube scan, which measures rule satisfaction rather than semantic correctness. The agreement of all three signals is consolidated into a three-way metric, the most concrete and verifiable measure of true remediation for RQ2.

Section 6.4 addresses RQ3, reporting per-agent processing time breakdowns and

hardware utilization metrics recorded on the `compai-01` server throughout the experiment.

6.1 Pipeline Remediation Outcomes

The pipeline processed all 100 sampled issues over approximately 16 hours on the `compai-01` server. Each issue passed through the entire pipeline, in particular the Summarizer, Remediation, Validation and Documentation agents, which use the Qwen3.5-27B open-weight model. The Remediation agent was permitted up to three attempts per issue before it was marked as failed. Through the pipeline's own checkpoint reporting, each issue was assigned one of three terminal outcomes:

Resolved The Remediation agent produced a fix that the Validation agent approved.

Failed The Remediation agent could not produce a fix the Validation agent would approve within the allowed attempts.

False Positive The Remediation agent judged the flagged code to be a non-issue, so no change was warranted.

As shown in Figure 6.1, 61 issues (61%) were reported as resolved, 34 issues (34%) failed and 5 issues (5%) were identified by the Remediation agent as false positives.

6.1.1 Outcome Breakdown

Figure 6.2 disaggregates outcomes simultaneously by severity and issue type, revealing patterns that neither dimension alone would expose. No Blocker-severity issues and no Vulnerability-type issues were present in the sample. By severity, the

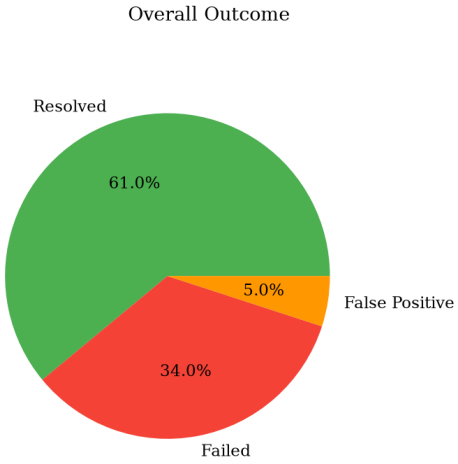

Figure 6.1: Pipeline reported overall outcome

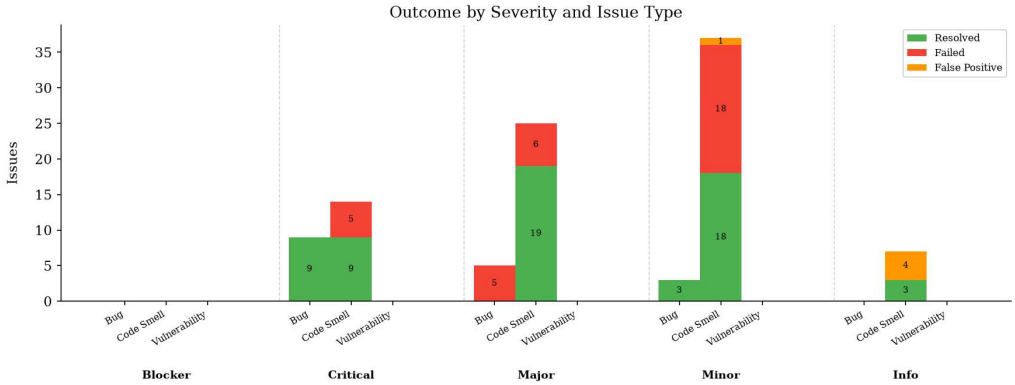

Figure 6.2: Pipeline reported outcome by severity and outcome by issue type

pipeline resolved 18 of 23 Critical issues (78%), 19 of 30 Major issues (63%) and 21 of 40 Minor issues (53%), a steady decline in resolution rate as severity decreases.

The cross-dimensional breakdown surfaces a more nuanced picture. At Critical severity, all nine Bug-type issues were resolved (100%), against nine of the fourteen Code Smells (64%). At Major severity the pattern reverses sharply, as all five Major Bugs failed (0%) while 19 of 25 Major Code Smells were resolved (76%). At Minor severity, all three Bugs were resolved, while the 37 Code Smells divided into 18 fixed, 18 failed and one false positive.

Manual inspection of the reasoning behind failed attempts surfaced two recurring failure modes. The first was token exhaustion, where the Remediation agent spent most of its token budget on extended reasoning and had too little left to produce an output block. The Validation agent's semantic reviewer occasionally failed the same way. The second was an implementation limitation, where arrow functions lacking a proper identifier were assigned a default marker of `<anonymous>`. When the Validation agent later tried to locate the function by name for patching and verification, the lookup failed and the attempt aborted.

At Info severity, all seven issues were Code Smells. Three of the seven (43%) were resolved and the remaining four were flagged as false positives, almost the entire false-positive count in the sample. Inspection of the reasoning showed that all four were `S1135` TODO comment violations, which the agent consistently deemed not to warrant a change.

6.2 LLM-as-Judge Evaluation

The pipeline's self-reported results only indicate whether the Validation agent accepted a fix. A more reliable measure of remediation quality needs an independent assessment made with the context of the full project. To this end, a post-hoc LLM-as-judge evaluation was conducted using Claude Sonnet 4.6 via Claude Code,

following the prompts in Appendix B. To keep the assessment unbiased, the judge was given full access to the Mermaid.js codebase and the sampled issues but was not told the system's own verdict. It assessed each issue across four dimensions:

Correctness Does the change resolve the reported SonarQube violation?

Semantic Preservation Does the change maintain the original observable behaviour of the code?

Regression Avoidance Does the change avoid introducing new issues into the codebase?

Idiomaticity Is the change minimal and consistent with the project's existing coding patterns?

Based on these dimensions, the judge assigned one of four labels: *correct*, *partially correct*, *incorrect* or *unsafe*, together with a confidence level of high, medium or low for its label assignment.

While the judge initially graded the 100 issues on their applied fixes, the false positives the Remediation agent had flagged were verified separately by the judge as well. Since triage is part of the remediation process, correctly declining a non-defect is itself a correct outcome, so a verified false positive is folded into the judge's *correct* label. On this basis, Table 6.1 presents the cross-tabulation of system outcome against judge label across the 100 sampled issues, with the four verified false positives counted as correct and the single overturned case as incorrect.

6.2.1 Judge Findings

The judge found 53 of the 61 pipeline-resolved issues correctly remediated. Folding in the four verified false positives gives 57 correct outcomes across the 100 sampled

Judge Label / System Outcome	Correct	Partially Correct	Unsafe	Incorrect	Total
Resolved	53	1	6	1	61
Failed	0	0	0	34	34
False Positive	4	0	0	1	5
Total	**57**	**1**	**6**	**36**	**100**

Table 6.1: Cross-tabulation of system outcome versus LLM-as-judge label across 100 sampled issues.

issues, a correct-handling rate of 57%. The eight resolved issues the judge did not accept were manually assessed and are examined below.

Six resolved issues were labelled *unsafe* by the judge, all at high confidence. Only one of these genuinely introduced a critical behavioural regression, while four shared a common limitation in the pipeline's AST-based function extraction. The context extraction captured the surrounding function block beginning from the identifier, omitting any preceding modifier keywords such as `export`. Since both the Remediation and Documentation agents produce an updated code block that is spliced back at the exact position of the extracted code, this inadvertently transferred the `export` keyword to the first extracted helper block or JSDoc comment, leaving the original function unexported.

The remaining discrepancies arose from the pipeline's fallback windowing strategy. Rather than passing the entire file to the Remediation agent, the pipeline works on a bounded window around the relevant code. While this reduces token usage and focuses the model, the fix may be applied without awareness of the wider context, producing changes that are syntactically valid yet semantically incomplete. Such cases arise where matching changes are also required elsewhere in the file. Several of the partially correct and incorrect verdicts, and one further unsafe verdict among the resolved issues, are attributable to this constraint.

6.3 SonarQube Post-Remediation Analysis

Following the pipeline run, all system-resolved changes were merged into a single branch and a post-remediation SonarQube scan was conducted on the full Mermaid.js codebase. This section reports the scan-confirmed resolution rate, quality delta and regression rate.

6.3.1 Scan-Confirmed Resolution Rate

Of the 61 issues the pipeline reported as resolved, 59 were confirmed closed by the post-remediation scan. Measured against the full 100-issue sample, this is a scan-confirmed remediation rate of 59%. However, the three-way agreement in Table 6.2, combining the pipeline's checkpoint reporting, the SonarQube post-scan and the LLM-as-judge verdict across all 100 issues, reveals a more accurate outcome.

Outcome	**Pipeline**	**SonarQube**	**Judge**	**Issues**
• = confirmed ○ = not confirmed				
Correctly remediated	•	•	•	53
Correctly declined (verified false positive)	○	○	•	4
Rule-satisfied but semantically incorrect	•	•	○	6
Pipeline incorrectly reported as resolved	•	○	○	2
Not resolved	○	○	○	35
Total				**100**

Table 6.2: Three-way agreement between the pipeline, SonarQube post-scan and LLM-as-judge verdicts across 100 sampled issues

The largest group, 53 issues, achieved full agreement across all three signals. This establishes a true remediation rate of 53%, the strictest measure of successful remediation across the 100 sampled issues. A complementary correct-handling rate of 57% additionally credits the four correctly declined false positives, reflecting the pipeline's capacity to triage issues correctly rather than only to remediate them. Of the five issues it flagged as false positives, four were genuine and one was overturned, a triage precision of 80%. These figures are reported separately, as remediation and

triage are distinct capabilities of the system that work together in sequence. Of the remainder, six issues were confirmed closed by both the agent and SonarQube but flagged by the judge (Section 6.2.1), satisfying the rule check while the underlying quality concern remained. Two more were marked resolved by the agent but confirmed by neither SonarQube nor the judge. The final 35 cases cover the 34 failed attempts and the one overturned false positive.

6.3.2 Regression Rate

Of the 44 files in which at least one issue was resolved, 11 (25%) exhibited net-new SonarQube issues in the post-remediation scan, introducing 26 new violations in total. Table 6.3 lists the affected files and the severity distribution of the introduced issues.

File	New Issues	Severity Breakdown
`materializedGeometry.ts`	6	Critical: 6
`ishikawaRenderer.ts`	4	Major: 4
`edges.js` (rendering-util)	4	Critical: 1, Major: 1, Minor: 2
`edges.js` (dagre-wrapper)	3	Critical: 2, Minor: 1
`stateRenderer.js`	2	Major: 2
`scoreLayout.ts`	2	Critical: 1, Major: 1
`classBox.ts`	1	Critical: 1
`utils.ts`	1	Major: 1
`index.js` (dagre-wrapper)	1	Major: 1
`shapeUtil.ts`	1	Minor: 1
`handle-markdown-text.ts`	1	Minor: 1
Total	**26**	

Table 6.3: Files with net-new SonarQube issues introduced by the pipeline, with severity breakdown.

The largest concentration of regressions, six new Critical violations, came from decomposing long functions into helpers or passing anonymous functions as arguments to built-in utilities. Both pushed cognitive complexity beyond SonarQube's default tolerance of 15.

6.3.3 Complexity and Other Quality Measures

Table 6.4 reports the project-wide quality and complexity metrics before and after remediation. The technical debt figures in this chapter are SonarQube's own estimated remediation effort in minutes [37]. They are a rule-based approximation and do not reflect real-world developer effort, which varies from developer to developer.

Metric	**Before**	**After**	**Delta**	**Change**
Cognitive Complexity	13,248	13,191	−57	−0.4%
Cyclomatic Complexity	23,499	23,546	+47	+0.2%
Code Smells	2,901	2,797	−104	−4%
Bugs	520	509	−11	−2%
Vulnerabilities	221	221	0	0%
Critical Violations	433	397	−36	−8%
Major Violations	1,522	1,491	−31	−2%
Minor Violations	1,534	1,491	−43	−3%
Technical Debt (min)	16,165	15,786	−379	−2%
Debt Ratio (%)	0.3	0.3	0.0	0%

Table 6.4: Project-wide SonarQube quality and complexity metrics before and after remediation.

At the project-wide level, the remediation produced small but measurable improvements across most quality dimensions. The codebase shed 104 Code Smells (−4%) and 11 Bugs (−2%), and its total technical debt fell by 379 minutes (−2%). The most pronounced gain was in Critical Violations, which fell by 36 (−8%), reflecting the pipeline's ability to dissect, understand and remediate affected code even at higher severity.

Two structural complexity metrics are also reported. Cyclomatic complexity counts the linearly independent paths through a program, adding one for each branching construct such as a conditional or loop. It is a common proxy for testability and maintainability. Cognitive complexity instead sums penalty points assigned to structures that make code harder to follow, such as nested conditionals and breaks in linear flow, with deeper nesting penalized more heavily. Both are plain counts of code structures and thus are dimensionless [37]. Furthermore, SonarQube re-

ports them as only project-wide metrics and thus their changes are relatively small. Here cyclomatic complexity rose marginally from 23,499 to 23,546, an increase of 47 (+0.2%), while cognitive complexity fell modestly from 13,248 to 13,191, a decrease of 57 (−0.4%). This is consistent with the function-extraction fixes redistributing nesting depth across smaller functions. Both movements are marginal against totals in the tens of thousands.

Table 6.5 narrows the analysis to the 100 sampled issues, the debt the pipeline was actually pointed at. Resolution here is scored by the strict three-way criterion rather than SonarQube closure alone, so the six rule-satisfied but judge-rejected fixes are treated as unresolved while the four verified false positives are credited. This is why the sample shows 57 correctly handled issues here, against the 59 that the SonarQube scan alone confirmed closed. The debt cleared therefore reflects only genuinely remediated issues, while the regression total counts every new violation the scan introduced. Weighted by SonarQube's effort estimate, the sample carried 792 minutes of debt. True remediation cleared 287 minutes of it, about 36%, while the 26 regressions added back 337 minutes. The net effect is a slight increase of 50 minutes (+6%), leaving the run close to break-even on a debt basis. On SonarQube's own ledger, which also credits the six disputed closures, the same sample shows a net reduction of about 6% instead.

The count and the debt diverge. The raw issue count over the sample fell by 31%, since the 57 correctly handled issues outnumber the 26 regressions introduced. The effort-weighted debt still rose, because those regressions were heavier, averaging about 13 minutes each against roughly 5 minutes for each issue the pipeline cleared. Eleven of the 26 new violations were Critical. This gap, a falling issue count alongside slightly rising debt, is the clearest quantitative argument for keeping the pipeline behind human review rather than auto-merging its output.

Metric	Before	After	Delta	Change
By Severity				
Critical	23	19	−4	−17%
Major	30	23	−7	−23%
Minor	40	27	−13	−33%
Info	7	0	−7	−100%
By Issue Type				
Bugs	17	8	−9	−53%
Code Smells	83	61	−22	−27%
Total Issues	100	69	−31	−31%
Technical Debt (min)	792	842	+50	+6%

Table 6.5: Quality delta across the 100 sampled issues. Resolved counts credit both true remediation and verified triage declines, and the after state includes the 26 regressions the pipeline introduced in the touched files.

6.4 Computational Resource Cost

Two complementary data sources determine the computational resource cost. The first is the per-issue timing logs recorded by the pipeline itself. The second is the system-level hardware utilization collected via Beszel [38] on `compai-01` throughout the experiment.

6.4.1 Processing Time

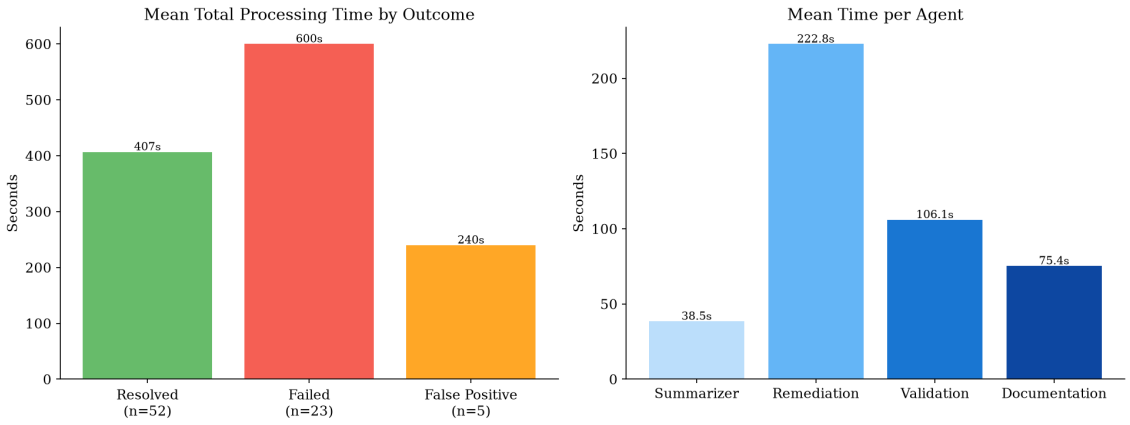

Figure 6.3: Processing time analysis. Total processing time by attempt number (left) and mean time per agent (right).

Figure 6.3 (left) presents processing time per unique function and code snippet, disaggregated by the three outcomes. Failed attempts took the longest at around 600 seconds on average, because they exhausted all retries. Successful remediation averaged around 407 seconds, while classifying an issue as a false positive took around 240 seconds.

Figure 6.3 (right) decomposes the timing across the four agents that have a language-model backend. The Summarizer, with its small task of extracting issue context, was the fastest at a mean of 38.5 seconds per issue. The Validation agent averaged 106.1 seconds and the Documentation agent 75.4 seconds. The Remediation agent took the longest at 222.8 seconds because it does two things. First, it determines whether a fix is actually necessary, which is how it flags false positives. Second, it performs the remediation itself, comprehending the issue, devising a fix strategy and generating the code change, and on retries it repeats all of this with the prior failure reasoning as context. Summing the mean agent times gives a rough per-issue estimate of about 442.8 seconds for a single-attempt resolution.

6.4.2 Hardware Resource Utilization

Table 6.6 summarizes the hardware utilization recorded throughout the experiment.

Metric	**Active (Peak)**
H100 NVL 0 GPU Utilisation	∼85%
H100 NVL 0 VRAM	∼40 GB
System CPU Utilisation	∼3.3%

Table 6.6: Hardware resource utilization

The Qwen3.5-27B model occupied roughly 40 GB of VRAM on a single H100 GPU with a total capacity of about 94 GB, roughly 43% of the available memory. This leaves ample headroom for a larger model or multiple models in future experiments. The GPU sustained around 85% utilization on average, while system-wide CPU utilization stayed near 3% during active runs, mostly from Ollama processes.

7 Discussion

This chapter interprets the findings of Chapter 6 against the four research questions that motivated the work, moving from measured outcomes to broader conclusions about how well an open-weight model remediates technical debt, what it costs to run locally and when the approach could realistically be adopted. The chapter opens with a short synthesis of the results, then takes each research question in turn. It closes with the study's limitations, the threats to its validity and the directions for future work.

7.1 Answering the Research Questions

Taken together, the results describe a working but still an early-stage prototype. This multi-agent pipeline, built entirely on a locally served open-weight model, ingested a static analysis report, remediated a sample of its issues and opened a reviewable pull request, all without any code leaving the host machine. By the strictest measure, the three-way agreement between the pipeline, an independent judge and a post-remediation SonarQube scan, 53 of the 100 sampled issues were genuinely resolved. That sits well below cloud frontier results, such as Khalil's roughly 90% code-level resolution with Claude Sonnet 4 in an agentic IDE [4] and the 98% refactoring precision at 52% recall of the ACE system [15]. But the prototype reached this result on a fraction of the resources, using one of the inference server's two GPUs, with no per-issue API cost and no code sent to a third party.

The subsections below interpret this against each research question in turn.

7.1.1 Designing a Multi-Agent Open-Weight Remediation System with Existing Tools and Workflows

RQ1 asked whether a multi-agent system built on open-weight models could be designed together with existing tools and workflows to remediate technical debt. The artifact conceptualized in Chapter 4 and realized in Chapter 5 answers this affirmatively. Packaged together to run as a container inside a GitHub Actions workflow, the pipeline integrates SonarQube as its detection layer, tree-sitter for language-aware code extraction and replacement, Git and the GitHub API for version control and Ollama for local model serving. The system consumes a standard static analysis report and produces a conventional pull request, which means it slots into a development team's existing review process rather than requiring a new one.

Beyond demonstrating that such a working system can be built, the design process surfaced several lessons about working within the constraints of a local open-weight model. Several decisions in the realized pipeline diverged from the proposed architecture precisely because of the model's limitations and these divergences are themselves findings about RQ1. The clearest example is the output format. The proposed design specified a unified diff for compactness, but the model could not reliably produce the precise positional arithmetic a diff demands, nor stop hallucinating lines when explicitly instructed not to.

This mirrors findings from a dedicated benchmark of diff understanding across model sizes. Glukhov et al. found a clear scaling trend on unified diff tasks, with smaller open-source models falling well behind larger ones, partly because they confused the format's single-character line markers with ordinary code content. The benchmark further showed that even as unified-diff performance improves with scale, it remains consistently outperformed by search-and-replace style formats [39]. This

supports the pipeline's approach. Rather than asking the model to generate a positional diff, remediation was split into full function-block generation by the model and placement via tree-sitter byte offsets, taking positional precision away from the model's generative uncertainty.

Two further divergences follow the same logic. The Validation agent was to reanalyse each patch with SonarQube and execute the project's tests, both of which proved impractical to run per patch and were replaced by a lightweight syntax check and a model-based semantic review. A triage step was also added, letting the model decline to change code it deemed already correct, saving further generation attempts.

The broader lesson for RQ1 is that the boundary between what is handled deterministically and what is delegated to the model must be drawn conservatively. Wherever the task required exact symbolic or character manipulation, tree-sitter and byte offsets carried it. Wherever it required judgement over code, the model did it. This division of labour is consistent with the multi-agent findings of [16] and with the observation that agentic sub-tasks are often simple and close-ended enough for compact models [5], and it is this design contribution that the thesis offers back to the knowledge base.

7.1.2 Effectiveness of Open-Weight Models in Agentic Technical Debt Remediation

RQ2 concerned how effectively open-weight models remediate technical debt, in terms of both remediation accuracy and code quality improvement. The headline result is the gap between what the pipeline reported and what could be independently verified. The system's own Validation agent marked 61 of 100 issues as resolved, but only 53 were confirmed as correct by all three evaluation layers. The discrepancy is not noise. It is a systematic overstatement produced by a validator that shares the generator's model, token budget and contextual scope. This is consistent with

evidence that language models struggle to reliably self-correct without an external signal, even when the generation and critique steps are separated into distinct calls [40], and it is the strongest methodological argument for the three-layer evaluation design. Had this thesis reported only the pipeline's self-assessment, it would have claimed a resolution rate that the developer inspecting the code would not agree with. The 53% after the three-way assessment is therefore the honest and true measure of remediation. Crediting the four false positives that the pipeline correctly declined, each verified by the independent judge, lifts the broader correct-handling rate to 57%, though the strict remediation figure stays at 53%.

When considering the model's performance alone, away from the pipeline's mechanical faults and logical errors, it is much better than what the aggregate rate suggests. Many of the failures reported were not just bad patches. The extraction defect and the unparseable responses removed issues before the model's reasoning could even be judged. When it did produce a complete output, that output was often correct. Its recurring weakness was not faulty logic but token capacity. On larger functions it often spent its entire token budget on reasoning and had none left to emit usable code. The main reason for this is the shift to whole function replacement, where regenerating a full function rather than a compact unified diff left very little room for the output.

The other half of RQ2 concerns patch quality. When the model returned a complete replacement, the code was usually idiomatic and syntactically valid. The weaknesses that surfaced were mostly subtle, such as helpers using variables from the original function without being passed in as parameters, or breaking a long function into helpers to reduce code complexity but failing to address the original issue. Each looked plausible and satisfied the rule check, yet introduced a problem elsewhere. Combined, these defects show up as a 25% file-level regression rate. Measured in SonarQube's estimated debt effort, the new issues offset much of the positive gains.

When compared against prior work, these figures seems modest but it is not discouraging. Frontier models have consistently produced higher numbers. For example, Khalil's experimentation reports roughly 90% code-level resolution using Claude Sonnet 4 [4]. However, that comparison is uneven in ways that matter. Khalil's fixes were produced inside an agentic IDE, where the model had access to the whole project and tools such as code editing and file navigation. By contrast, the system here drives a single open-weight model over a bounded window of code with none of those affordances. Furthermore, the codebase was mid-sized proprietary work which did not go through an independent three-way verification of the kind applied here. A fairer reading is that a 27B open-weight model, which is lean in comparison, running locally, recovers a little over half of the technical debt a static analyzer flags from a large-scale open-source project using only a limited view of its surrounding context. Set against open-weight results on a related SATD-repair task, where exact-match resolution for comparable models sat in the low tens of percent [14], a verified 53% on general code-level debt resolution is a meaningful step forward. This brings empirical evidence to the thesis that local open-weight models are becoming a viable option for software maintenance especially with human overight.

7.1.3 Computational Resource Requirements and Operational Costs of Open-Weight Remediation

RQ3 asked what the computational and operational costs of open-weight model-based remediation are. On hardware, the answer is encouraging. Configured with a 32,768-token context window and an 8,192-token response cap, the Qwen3.5-27B model ran on just one of the server's two H100 NVL GPUs, occupying roughly 40 GB of that GPU's 94 GB during the run. That is about 43% of its capacity with around 85% utilization. While the system CPU was barely used throughout the

whole run, the second GPU was left completely unused. This suggests that, even though the experiment ran on a datacentre-grade H100, its consumption places the approach well within reach of an organization self-hosting on more modest hardware. Crucially, this is a fixed, one-time hardware cost with no per-issue charge and no data egress.

Time is the one cost that is genuinely significant. A single-attempt resolution took around seven to eight minutes of model time, whereas failed issues that exhausted all three retry attempts averaged around ten minutes. Including all other inference, the full 100-issue sample took roughly sixteen hours. For an interactive use case, this would be extremely limiting. However, the system is designed to run asynchronously as a CI (Continuous Integration) batch job, clearing an accumulated backlog overnight or over the weekend and presenting a pull request the next day. For such a pattern this cost is less significant. As a prototype, it also leaves room for optimization. The triage step, for instance, is re-run on every retry even though its verdict never changes. The lightweight Summarizer and Documentation agents could run on a smaller, distilled or more aggressively quantized model, both of which would cut time further (Section 7.3). Considering all of this, for asynchronous batch remediation, the operational cost is well matched to the workload and has clear room to fall even further.

7.1.4 Feasibility of Integrating and Deploying Local Open-Weight Remediation in Real-World Development Workflows

RQ4 asked how feasible it is to integrate and deploy local open-weight remediation in a real-world development workflow and how that compares to a generalist cloud-based LLM for wider adoption. Feasibility, in this case, is not a single property but what emerges when capability, cost, privacy and fit are weighed together.

On raw numbers, the open-weight model remains behind cloud frontier models. But as RQ2 established, it still resolved just over half of the sampled issues on a large open-source codebase, and it did so without the whole-project access a frontier model relies on. That constraint should change how a development team reads the number.

The preceding subsection also established the hardware cost. But a more relevant comparison is with the recurring price of a cloud API. As of writing, Table 7.1 lists the published per-million-token rates of current flagship models from the three major providers, in standard and batch modes [41], [42], [43]. Because remediation is asynchronous, a cloud deployment could use the batch tier and halve these rates. Even so, the pipeline makes several calls per issue, for summarization, triage, remediation, validation and documentation, along with any retries for failed attempts. With a frontier cloud model, that per-token charge scales with the backlog, the runtime and the number of projects. Self-hosting converts it into a fixed upfront cost that one inference server amortizes across an organization's many repositories, at the price of maintaining the server itself. Whether that trade-off is acceptable depends on the next concern.

		Standard		**Batch**	
Provider	**Model**	**Input**	**Output**	**Input**	**Output**
OpenAI	gpt-5.6-sol	5.00	30.00	2.50	15.00
	gpt-5.6-terra	2.50	15.00	1.25	7.50
	gpt-5.6-luna	1.00	6.00	0.50	3.00
Anthropic	Claude Fable 5	10.00	50.00	5.00	25.00
	Claude Opus 4.8	5.00	25.00	2.50	12.50
	Claude Sonnet 5	3.00	15.00	1.50	7.50
Google	Gemini 3.6 Flash	1.50	7.50	0.75	3.75
	Gemini 3.5 Flash	1.50	9.00	0.75	4.50

Table 7.1: Published frontier proprietary model API pricing (USD per 1M tokens).

The stronger case for local deployment is privacy. Because the model is served on the organization's own infrastructure, no source code, intermediate artifact or

generated patch ever leaves it. For teams in regulated sectors such as finance, healthcare, defence or public administration, where sending proprietary code to a third party is restricted or outright prohibited, this is not a convenience but a precondition, and it is the one requirement a cloud model cannot meet at any price.

Combined with the findings of RQ1, this shows that integration itself is low-friction. The system runs as a CI job within an existing CI/CD pipeline and delivers its output as an ordinary pull request, so developers need no new tools or process and the fixes arrive in the usual review path they already use. Deployment is where the choice divides, though it increasingly favours the local option. On one side, an organization must procure and maintain its own GPU-backed inference server, which for most is an easy call once privacy and hardware footprint are factored in. On the other, a cloud provider removes that burden for a recurring API fee and the requirement to grant access to the codebase.

7.2 Limitations

Even though the research and experimentation made some novel contributions, there are several limitations that bind the strength of its conclusions. The most fundamental is scope, a consequence of the limited time available for this research. The evaluation covers a single codebase, albeit a large-scale publicly available open-source project, Mermaid.js, a single language family, TypeScript and JavaScript, and a single open-weight language model, Qwen3.5-27B. The findings should therefore be read as a detailed case study rather than a general characterization of open-weight model-based remediation as the resolution rates in particular may not transfer to other languages, project styles or models.

The sample size also limits some of the more detailed claims. One hundred issues drawn from a filtered pool of 1,460 is enough for some overall rates but leaves the finer categories very small. The five Major Bug issues, for instance, all failed and

the percentages from such small groups are easily skewed by a single case. The quality gains are similarly hard to read, since they appear only marginal against project-wide metrics. Figures at this level of detail should therefore not be read too closely.

A further limitation lies in the scope of the context bundle. The bundle was designed to provide context from the file being fixed, but gave the pipeline no visibility of where a function is used across the wider project. A function referenced from other files cannot be changed safely without updating those call sites as well. Yet the pipeline could neither see nor modify them. So any change with cross-file impact could not be carried through consistently. This is one source of the regressions noted earlier.

Two aspects of the inference configuration were left unexplored, both affecting the token exhaustion behind many failures. First, the token budget was set to a 32,768-token context window and an 8,192-token response cap. This is generous for a typical function, but the project held many functions long enough to exhaust the cap before the rewrite could finish. Thus, it is unclear whether a larger cap on either or both of these variables would have reduced the truncated outputs. This also makes Mermaid.js atypical. A project with more typically sized functions might have produced higher resolution rates with the same configuration, giving a more generalized understanding of the approach's effectiveness.

Second, prompts to limit the model's chain-of-thought reasoning were not tried. The Remediation agent needs that reasoning to produce a working fix, yet it shares that same budget with the code it generates, so reasoning less could leave more room for the output. On the other hand, the Summarizer and Documentation agents need none at all. Both changes could have shortened processing time, though their effect on fix quality is left to future work.

The evaluation instrument carries its own caveats. Most specifically, the judge

was run through Claude Code with Claude Sonnet 4.6 under its default configuration. So the underlying system prompt was set by the tool rather than by the author. This means reproducing the judging step exactly would require the same tool version and settings. LLM-as-judge assessments also carry known biases and can misclassify incorrect code or hallucinate faults even in strong judges [27], [44]. However, the three-layer design was built to contain this by comparing the judge's verdicts against the pipeline's own findings and a deterministic SonarQube scan rather than relying on the LLM's decisions alone. The false-positive check was small, covering only the five issues the pipeline declined, so its triage precision is a rough estimate.

Finally, an early implementation defect surfaced during the run. A flaw in how functions were extracted and spliced back left some fixed functions no longer exported, which both the syntax check and the SonarQube scan missed. it was only the independent judge that caught this subtelty, marking four fixes that suffered the same issue. Though the defect was fixed afterwards, a re-run of the experiment was not conducted due to time constraints. So the reported results reflect the pipeline as it behaved at the time.

7.3 Future Direction

Perhaps the most striking result of this work is that even open-weight models running on local hardware can meaningfully remediate real technical debt. This was shown on a single large-scale open-source project, so the clearest next step is breadth. Repeating the study across more codebases, languages and open-weight models would show how far the approach generalizes. Similarly, a study with practising developers reviewing the generated pull requests would add another dimension to the evaluation framework, testing the judgement of an LLM against human review. At the same time, it could measure how much effort the system realistically saves.

The system itself also has clear room to improve, beginning with how models are

assigned to agents. While the current implementation ran a single model for all of its agents, serving different specialized open-weight models to different agents could match each task to a model of the right size and skill. The lightweight Summarizer and Documentation agents could run on a small, fast model, while the Remediation and Validation agents, which carry the reasoning load, could use a larger or even a smaller but specialized one. This was deliberately deferred in the model selection (Chapter 3) to keep inter-model variability out of the results. But it would be the natural next step to establish how a multi-model assignment performs in remediation.

Validation is a second candidate for change. At present a single reviewer checks each fix with the same model as every other agent in the system. So it shares their blind spots as well. Thus, in addition to the proposal on different specialized models earlier, several validators, or more specifically a multi-judge consensus-based validation, could be implemented. This could ideally make the pipeline's validation more self-reliant and narrow the gap between what the pipeline reports and what is truly resolved. Furthermore, proper support for running the project's tests or using a test-generating agent would also add back the behavioural check that was missing in this implementation.

A further set of directions share the same idea. Each equips the agents with tools to fetch what they need during a run rather than relying on inputs prepared in advance. The MCP (Model Context Protocol) is an open-source protocol that provides a standard way to expose such tools to a language model [45], and two parts of the pipeline could use it. The first is context building. In place of the static context built from the single file the target function sits in, a tool-using context agent could decide for itself what related code to pull in from across the project. This would address the cross-file problems behind several failures and let the Remediation agent consider the wider impact before it makes any changes.

The second is detection. This research used SonarQube as its only source of issues, but additional analysers such as Snyk [12] or CodeScene [17], for which Tornhill et al. built ACE [15], could be added behind the same MCP interface to widen the range of debt the system acts on. Additionally, the validation agent could use the same interface to re-check a fix against an analyser during the run instead of relying only on the report gathered beforehand. This would reintroduce the runtime dependency the report-based design deliberately removed, but in return it could reduce failed attempts.

Finally, as an alternative focus to the remediation system itself, this work leaves open the question of making unified diff generation viable with open-weight models. Switching to full function replacement worked around the model's inability to produce correct diffs, but it did so with far larger outputs, which drove much of the token exhaustion. A model fine-tuned or prompted specifically to emit reliable diffs or a constrained decoding scheme that enforces valid diff structure, could restore the compactness of the original design without giving up the correctness that full replacement brought. Reliable diff generation from open-weight models would benefit not just this pipeline but any local language model based system where output size is a direct driver of cost.

8 Conclusion

Modern frontier large language models have shown extraordinary capabilities across many general-purpose tasks. In software development and maintenance, they have become the backbone of agentic workflows, powering tools that write, review and repair code. But their growing capability raises questions of privacy and ownership, and of whether the rules and regulations governing sensitive data are respected. Frontier model providers often neglect these concerns. And even when they do offer options to opt out of data retention, it is often unclear, hard to find or locked behind a paid tier. For teams building and maintaining data-sensitive systems, such as those in healthcare, finance, defence or critical infrastructure, keeping code and data in-house is a hard requirement that rules out most cloud models altogether. In recent years, however, open-weight language models have shown real promise, often catching up to newer frontier releases. Similarly, the rise of specialized small language models, together with compression techniques like quantization and distillation, has lowered the hardware needed to run them. This has made self-hosting practical for far more organizations and driven the adoption of open-weight models.

Motivated by this and by prior works, this thesis set out to explore the practical capabilities of open-weight models in automated technical debt remediation, with four research questions as its pillars. In answering them, it presents both a design and a working artifact of a multi-agent pipeline that runs entirely on such a model and delivers its fixes as a reviewable pull request, keeping the development team

in control of every change. The system was evaluated on a stratified sample of 100 SonarQube issues from Mermaid.js, a large-scale open-source project, using a three-layer evaluation that combines the pipeline's own verdict, an independent LLM-as-judge and a post-remediation static analysis scan. Each of these questions is revisited below.

RQ1 *How can a multi-agent open-weight system be designed with existing tools and workflows?* The realized pipeline uses SonarQube for detection, tree-sitter for language-aware code extraction and replacement, Git and the GitHub API for version control and Ollama for local model serving. Packaged as a container to run inside a GitHub Actions workflow, it consumes a standard static analysis report and produces a conventional pull request, slotting into a team's existing review process. The central design lesson is to find the right balance between generation and logical procedure. Work that requires deterministic results is much better off if done by existing libraries and tooling, whereas work requiring judgement is better left to a generative model.

RQ2 *How effectively can open-weight models remediate technical debt?* The open-weight model remediated technical debt with modest but meaningful effectiveness. When considering the three-way verdicts from the validation agent, the SonarQube post-scan and an independent LLM judge, 53% comes out as the honest measure of true remediation. This sits below the rates reported for cloud frontier approaches but above comparable open-weight results on related repair tasks. Furthermore, it reached these results using a model that is on the smaller side, without the whole-project access that most systems with a frontier model have. When the four false positives the pipeline correctly discerned are also credited, the correct-handling rate rises to 57%. The changes were not always clean, but the 25% file-level regression rate this produced only shows that agentic remediation works best under human review.

RQ3 *What are the computational and operational costs?* The computational and operational cost is well suited to self-hosting. The 27B model sits at the smaller end of the model family and ran on a single H100 GPU in a 4-bit quantized form, using roughly 43% of its VRAM at around 85% utilization. This puts the approach well within the reach of an organization running its own inference server rather than only a large datacentre. It is a fixed, one-time hardware cost rather than the recurring bill of a cloud provider, with no per-issue charge and no data leaving the organization. The one real caveat is time. At roughly seven to eight minutes per issue it rules out interactive use, which is fine for a system that was built for an asynchronous, backlog-clearing batch job.

RQ4 *How feasible is it to integrate and deploy local open-weight remediation, and how does it compare to cloud models?* Local open-weight remediation is feasible now as a privacy-preserving, review-gated backlog assistant. It remains behind cloud frontier models on raw capability, but on cost, privacy and workflow fit it is a credible alternative. Integration is low-effort as the design allows plugging into an existing CI/CD workflow and fixes arrive as ordinary pull requests. So developers need no change to their own workflow. As for deployability, it is straightforward thanks to Ollama and similar runtimes. But the up-front hardware cost of an in-house inference server can make a cloud API the easier short-term choice. In the long run, however, local inference wins out, both for organizations in regulated sectors that cannot send code to a third party and for teams wary of a recurring API bill.

In answering these questions, the thesis makes four contributions. First, it delivers a privacy-preserving multi-agent remediation system that operates entirely on locally served open-weight models and integrates with standard developer tooling. Second, it offers design knowledge on how to build agentic systems around open-weight models. Third, it proposes a three-layer evaluation methodology that

separates genuine, developer-grounded correctness from mere satisfaction of a static analysis rule. Finally, it contributes empirical evidence on the accuracy, cost and feasibility of remediation with local open-weight models.

The broader significance of this work is that a capable open-weight model, running even on a single self-hosted GPU, can meaningfully remediate real technical debt without any code leaving the organization. This is not yet a replacement for a frontier cloud model on raw capability, but it is a practical and immediately usable option for teams whose privacy or cost constraints rule out the cloud. As open-weight models continue to improve, the balance is likely to shift further in favour of the local approach. The most valuable next steps, detailed in Chapter 7, are to refine and test the concept across more languages, projects and models and to equip the agents with tools to gather context as they see fit.

References

[1] P. Avgeriou et al., "Technical debt management: The road ahead for successful software delivery", in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, IEEE, 2023, pp. 15–30. DOI: 10.1109/ICSE-FoSE59343.2023.00007.

[2] T. Besker, A. Martini, and J. Bosch, "Software developer productivity loss due to technical debt—a replication and extension study examining developers' development work", *J. Syst. Softw.*, vol. 156, pp. 41–61, 2019. DOI: 10.1016/j.jss.2019.06.004.

[3] A. Tornhill, "Calculate the costs of technical debt to optimize your software delivery", CodeScene, White paper. Accessed: Jul. 29, 2026. [Online]. Available: https://codescene.com/hubfs/calculate-business-costs-of-technical-debt.pdf.

[4] R. Khalil, "Design and evaluation of an AI-driven workflow for technical debt remediation in software systems", M.S. thesis, University of Turku, Finland, 2025. [Online]. Available: https://urn.fi/URN:NBN:fi-fe2025073080111.

[5] P. Belcak et al., "Small language models are the future of agentic AI", 2025. eprint: 2506.02153.

[6] S. Teki, *Slash agentic AI costs by 10-30x: The shift to small language models (SLMs)*. Accessed: Oct. 31, 2025. [Online]. Available: https://www.sundeepteki.org/blog/small-language-models-for-agentic-ai.

[7] SonarSource SA, *Sonarqube*. Accessed: Apr. 16, 2026. [Online]. Available: https://www.sonarqube.org/.

[8] Cursor, *Cursor*, 2026. Accessed: Jul. 28, 2026. [Online]. Available: https://cursor.com/get-started.

[9] Lovable, *Lovable*, 2026. Accessed: Jul. 28, 2026. [Online]. Available: https://lovable.dev/.

[10] Anthropic, *Claude code*, 2026. Accessed: Jul. 28, 2026. [Online]. Available: https://claude.com/product/claude-code.

[11] S. B. Pandi, S. A. Binta, and S. Kaushal, "Artificial intelligence for technical debt management in software development", 2023. eprint: 2306.10194.

[12] *What's snyk?* Accessed: Jul. 25, 2026. [Online]. Available: https://docs.snyk.io/whats-snyk.

[13] F. Zampetti, A. Serebrenik, and M. Di Penta, "Automatically learning patterns for self-admitted technical debt removal", in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2020. DOI: 10.1109/SANER48275.2020.9054868.

[14] M. S. Sheikhaei, Y. Tian, S. Wang, and B. Xu, "Understanding the effectiveness of LLMs in automated self-admitted technical debt repayment", 2025. eprint: 2501.09888.

[15] A. Tornhill, M. Borg, N. Hagatulah, and E. Söderberg, "ACE: Automated technical debt remediation with validated large language model refactorings", 2025. eprint: 2507.03536.

[16] S. Bakane, "Designing agentic AI for autonomous risk triage and false positive reduction in static security analysis", M.S. thesis, University of Turku, Finland, 2025. [Online]. Available: https://urn.fi/URN:NBN:fi-fe2025091195578.

[17] CodeScene AB, *CodeScene documentation*. Accessed: Jul. 25, 2026. [Online]. Available: https://codescene.io/docs/.

[18] Anomaly, *Opencode documentation*, 2026. Accessed: Jul. 28, 2026. [Online]. Available: https://opencode.ai/docs/.

[19] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research", *Manag. Inf. Syst. Q.*, vol. 28, no. 1, pp. 75–106, 2004. DOI: 10.2307/25148625.

[20] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "The technical debt dataset", in *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, New York, NY, USA: ACM, 2019. DOI: 10.1145/3345629.3345630.

[21] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies", in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, IEEE, 2021, pp. 560–564. DOI: 10.1109/MSR52588.2021.00074.

[22] K. Sveidqvist and Contributors to Mermaid, *Mermaid: Generate diagrams from markdown-like text*, Dec. 2014. [Online]. Available: https://github.com/mermaid-js/mermaid.

[23] Mermaid, *Integrations*. Accessed: Jun. 8, 2026. [Online]. Available: https://mermaid.js.org/ecosystem/integrations-community.html.

[24] Arena, *Code arena*. Accessed: Apr. 16, 2026. [Online]. Available: https://arena.ai/leaderboard/code?license=open-source.

[25] *Open LLM leaderboard*. Accessed: Apr. 16, 2026. [Online]. Available: https://llm-stats.com/leaderboards/open-llm-leaderboard.

[26] T. Dettmers and L. Zettlemoyer, "The case for 4-bit precision: K-bit inference scaling laws", 2022. eprint: 2212.09720.

[27] L. Zheng et al., "Judging LLM-as-a-judge with MT-bench and chatbot arena", 2023. eprint: 2306.05685.

[28] J. Liu et al., "Large language model-based agents for software engineering: A survey", 2024. eprint: 2409.02977.

[29] *LangGraph overview*. Accessed: Jul. 10, 2026. [Online]. Available: https://docs.langchain.com/oss/python/langgraph/overview.

[30] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge, England: Cambridge University Press, 2017. DOI: 10.1017/9781316771273.

[31] Tree-sitter, *Tree-sitter documentation*, 2026. Accessed: Jul. 19, 2026. [Online]. Available: https://tree-sitter.github.io/tree-sitter/.

[32] Ollama, *Openai compatibility*, 2025. Accessed: Jul. 21, 2026. [Online]. Available: https://docs.ollama.com/api/openai-compatibility.

[33] Qwen Team, *Qwen3.5-27b model card*, 2026. [Online]. Available: https://huggingface.co/Qwen/Qwen3.5-27B.

[34] M. Shen et al., "Optimizing LLM-based multi-agent system with textual feedback: A case study on software development", 2025. eprint: 2505.16086.

[35] S. Schulhoff et al., "The prompt report: A systematic survey of prompt engineering techniques", 2024. eprint: 2406.06608.

[36] A. Kong et al., "Better zero-shot reasoning with role-play prompting", 2023. eprint: 2308.07702.

[37] SonarSource, *Understanding measures and metrics*, n.d. Accessed: Jul. 29, 2026. [Online]. Available: https://docs.sonarsource.com/sonarqube-server/latest/user-guide/code-metrics/metrics-definition/.

[38] Beszel, *Beszel: Simple, lightweight server monitoring*. Accessed: Jun. 26, 2026. [Online]. Available: https://beszel.dev/.

[39] E. Glukhov, M. Conti, E. Bogomolov, Y. Golubev, and A. Bezzubov, "Diff-XYZ: A benchmark for evaluating diff understanding", 2025. eprint: 2510.12487.

[40] J. Huang et al., "Large language models cannot self-correct reasoning yet", 2023. eprint: 2310.01798.

[41] OpenAI, *Pricing*. Accessed: Jul. 23, 2026. [Online]. Available: https://developers.openai.com/api/docs/pricing.

[42] Anthropic, *Pricing*. Accessed: Jul. 23, 2026. [Online]. Available: https://platform.claude.com/docs/en/about-claude/pricing.

[43] *Gemini developer API pricing*. Accessed: Jul. 23, 2026. [Online]. Available: https://ai.google.dev/gemini-api/docs/pricing.

[44] C. B. Ozer, *Utilising LLM-as-a-Judge to evaluate LLM-generated code*, Jan. 2026. Accessed: Jun. 12, 2026. [Online]. Available: https://medium.com/softtechas/utilising-llm-as-a-judge-to-evaluate-llm-generated-code-451e9631c713.

[45] *What is the model context protocol (MCP)?*, Jun. 2026. Accessed: Jul. 25, 2026. [Online]. Available: https://modelcontextprotocol.io/docs/getting-started/intro.

Appendix A Agent Prompt Templates

This appendix collects the prompt templates used by the pipeline. Only the agents that call the language model are listed here. Thus, the Orchestrator, Context and VCS agents, which are deterministic and use no prompts, do not appear here.

Each template is reproduced as it exists in the source. The text in braces, such as `{language}` or `{issues}`, marks a placeholder that is filled in at runtime before the prompt is sent to the model. Every agent is given a system prompt that fixes its role and a user prompt that carries the task and the code. An agent that carries out more than one sub-task has a separate template for each. All of these are presented as well.

A.1 Summarizer Agent

The Summarizer agent produces a concise behavioural description of a target function. This description is later passed to the Remediation agent as a behavioural contract that the fix must maintain.

Listing A.1: Summarizer agent system prompt.

```
You are a {language} code analyst. Describe the given function
concisely. Respond ONLY with valid JSON - no prose, no markdown
```

```
    fences.
```

Listing A.2: Summarizer agent user prompt.

```
Describe the following function.


Return a JSON object with this exact shape:
{
  "description": "<one or two sentences explaining what the
   function does>",
  "parameters": [
    {"name": "<param name>", "type": "<type or 'unknown'>", "
   purpose": "<brief purpose>"}
  ],
  "returns": "<what the function returns, or 'void' / 'None'>"
}


Function source:
{source}
```

A.2 Remediation Agent

The Remediation agent works in two stages. It first triages an issue to decide whether a code change is genuinely needed. If a change is needed, it then produces a patch. Two patch templates exist, one for whole functions and one for fallback code snippets.

Listing A.3: Remediation agent triage system prompt.

```
You are a {language} refactoring expert. Respond ONLY with
   valid JSON
- no prose, no markdown fences.
```

Listing A.4: Remediation agent triage user prompt.

```
The following SonarQube issues were found in a {language}
   function.
Decide whether a code change is required to resolve them.

Issues:
{issues}

Code:
{source}

Return a JSON object:
{"fix_needed": true|false, "reason": "<brief explanation>"}
```

Listing A.5: Remediation agent patch system prompt.

```
You are a {language} refactoring expert. Respond ONLY with
   valid JSON
- no prose, no markdown fences.
```

Listing A.6: Remediation agent function patch user prompt.

```
Fix the SonarQube issues listed below without changing the
   observable
behaviour described in the summary.
```

```
File: {file_path}
Function: `{fn_name}` (starts at line {fn_start})

Issues:
{issues}

Behavioural contract (DO NOT break this):
{summary}
{rejection_block}
Existing imports (do NOT add any import/using/require not
   already present):
{import_block}

Existing function signatures (do NOT reuse these names for new
   helpers):
{name_inventory}

Constraints:
- Do NOT change any public method or function signature.
- Return the complete, fixed source of `{fn_name}` only - no
   imports,
  no class wrapper, just the function.
- If you extract helper functions, return their complete source
    in the
  helpers array.
- Before returning, audit every helper: every variable it uses
   must be
  declared locally, passed as an explicit parameter, or be a
```

```
   module-level
  import or global. Do NOT implicitly capture variables from
   the calling
  function's scope - pass them explicitly as parameters.

Current function:
```
{fn_source}
```

Return a JSON object with this exact shape:
{
  "replacement": "<complete fixed source of `{fn_name}`>",
  "helpers": [
    {"name": "<helper name>", "source": "<complete helper
   source>"}
  ]
}
If no helpers are needed, return an empty array for helpers.
```

Listing A.7: Remediation agent snippet patch user prompt.

```
Fix the SonarQube issues listed below in the module-level code
   shown.

File: {file_path}
Lines: {fn_start}-{fn_end}

Issues:
```

```
{issues}
{rejection_block}
Existing imports (do NOT add any import/using/require not
   already present):
{import_block}

Constraints:
- Do NOT wrap the code in a function or class.
- Return only the fixed version of exactly the lines shown.
- Do NOT add new imports.

Current code (lines {fn_start}-{fn_end}):
```
{fn_source}
```

Return a JSON object:
{"replacement": "<fixed version of the shown lines>"}
```

A.3 Validation Agent

The Validation agent checks whether a patch fully resolves the targeted issues without introducing regressions. Its verdict determines if the code changes should be applied to disk.

Listing A.8: Validation agent system prompt.

```
You are a code review expert. Respond ONLY with valid JSON - no
    prose,
```

```
no markdown fences.
```

Listing A.9: Validation agent user prompt.

```
Review whether the patched code fully resolves the SonarQube
   issues
listed below without introducing regressions.


Issues that must be resolved:
{issues}


Patched code:
```
{patched_code}
```


Return a JSON object:
{"passed": true|false, "reason": "<concise explanation>"}
```

A.4 Documentation Agent

The Documentation agent documents the fixed code and writes a conventional-commit message summarizing the fix. It reuses the Summarizer agent description for the target function, so only the helper functions the Remediation agent extracted need a fresh docstring. The prompts below cover that helper docstring and the commit message.

Listing A.10: Documentation agent docstring system prompt.

```
You are a {language} documentation expert. Respond ONLY with
   valid JSON
- no prose, no markdown fences.
```

Listing A.11: Documentation agent docstring user prompt.

```
Generate a docstring for this {language} function.
Return JSON: {"description": "...", "parameters": [...], "
   returns": "..."}

Function:
{source}
```

Listing A.12: Documentation agent commit message system prompt.

```
You are a software engineer. You produce concise conventional-
   commit messages. No prose.
```

Listing A.13: Documentation agent commit message user prompt.

```
Write a single conventional-commit message for this fix.
Format: fix(<filename>): <one sentence>

File: {filename}
Issues resolved: {issues}
Function summary: {description}
```

Appendix B LLM-as-Judge Prompts

The following prompts are given to the independent LLM-as-judge to assess both the code changes and the triage decisions made by the pipeline. The judge uses Claude Sonnet 4.6 via the Claude Code extension in VS Code with full access to the codebase and tooling.

B.1 Per-Issue Assessment

The judge assesses each sampled issue in two steps. First, it studies every issue including any related code, taking notes while it is at it. Next, it reads the diff between the fix branch and the base branch and labels each fix according to the criteria.

Listing B.1: LLM-as-judge per-issue preparation prompt.

```
You are a typescript-analyzer. I will give you a JSON file
   containing sampled SonarQube issues. For each issue, locate
   the flagged code at the specified file and line, understand
   what the surrounding function does, why SonarQube flagged it
    and what a correct fix would look like. Take notes per
   issue as you will use these to judge fixes shortly. Do NOT
   look at any fixes yet. Do NOT make any changes to the
   codebase.
```

Listing B.2: LLM-as-judge per-issue assessment prompt.

```
The fixed branch is <fixed branch>
The main branch is <main branch>
You are allowed to use git diff.

For each of the issues listed in the JSON, compare the actual
   fix in the diff against your pre-fix understanding of the
   code. Assess each fix on:

Correctness: does it resolve the SonarQube violation?
Semantic preservation: does it preserve the original behaviour?
No regression: does it avoid introducing new issues?
Idiomaticity: is the fix minimal and consistent with the
   project's existing patterns?
Acceptable: Are the changes acceptable.

Write your answers to a jsonl file.

Assign one label per issue: correct / partially correct /
   incorrect / unsafe.
Also assign confidence: high / medium / low.
```

B.2 False Positive Detection

Similar to the per-issue assessment, the judge also reviews the issues the pipeline declined as false positives in two steps.

Listing B.3: LLM-as-judge false-positive preparation prompt.

```
You are a typescript-analyzer. I will give you a JSON file
   containing sampled SonarQube issues. For each issue, locate
   the flagged code at the specified file and line, understand
   what the surrounding function does, why SonarQube flagged it
    and what a correct fix would look like. Take notes per
   issue as you will use these to judge false positive verdicts
   . Do NOT look at any fixes yet. Do NOT make any changes to
   the codebase.
```

Listing B.4: LLM-as-judge false-positive verdict prompt.

```
Given the list of issue ids, determine if a code fix is
   required. Return a jsonl file with the JSON structure {id:
   "", verdict: "won't fix" or "fix", reason: "simple
   explanation."}
```