Thesis/Report submitted to the

UNIVERSITY OF MOHAMED BOUDIAF - M'SILA, ALGERIA



FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

in partial fulfillment of the requirements for the degree of

Master's in Computer Science

Specialization: Computer Science

Option: ISSE

Presented by

BENNIA Souleymane Karim
BENYOUNES Mehdi

Entitled

AI-driven observability platform for application reliability

Under the supervision of

DEBBI Hichem

Jury Members

Dr. Noureddine AMRAOUI	University of M'sila	President
Prof. Hichem DEBBI	University of M'sila	Supervisor
Dr. Seddik BOUGHERARA	University of M'sila	Examiner

June, 2026

الملخص

في السنوات الأخيرة، شهد تطوير البرمجيات المدعوم بالذكاء الاصطناعي اعتماداً واسعاً، مما زاد من إنتاج الكود البرمجي وسرّع دورات التسليم. ورغم ما أحدثه من تحسين في إنتاجية المطورين، إلا أنه يطرح تحديات جديدة في مراجعة الكود وضمان الجودة وموثوقية التشغيل. تتعامل الحلول الحالية مع مراحل منعزلة من دورة حياة البرمجيات، دون وجود اتصال مستمر بين التحليل قبل الدمج والاستجابة للحوادث بعد النشر. يدفع هذا التشتت إلى الحاجة لمنصة موحّدة تدمج مراجعة الكود ورصد الإنتاج في سير عمل واحد. يقدّم هذا المشروع Falcon، منصة رصد تعمل بالذكاء الاصطناعي صُمّمت لسدّ هذه الفجوة عبر توحيد عمليات ما قبل الدمج وما بعد النشر. يدمج النظام محرك رسوم بيانية للتبعيات لتحليل الكود الثابت باستخدام ts-morph، وبيئة تنفيذ معزولة لاستكشاف الكود بشكل آمن، وتكامل مع GitHub، واستدلالاً وكيلياً للمراجعة المنظمة للكود والتوليد التلقائي للإصلاحات. يتيح النظام كشف العيوب عبر الملفات أثناء تحليل Request Pull وتقصّي الأسباب الجذرية لأعطال التشغيل عبر التكامل مع Sentry، والتوليد الآلي للتصحيحات مع التحقق من سلامة الأنواع. توضّح دراسة حالة أُجريت على مستودع إنتاجي بلغة TypeScript جدوى توحيد مراجعة الكود ومعالجة الحوادث ضمن سير عمل واحد مدعوم بالذكاء الاصطناعي. من خلال الجمع بين تحليل المستودع ورصد زمن التشغيل والاستدلال الوكيلي، يدعم Falcon ضمان الجودة المستمر للبرمجيات ويساهم في تحسين موثوقية وقابلية صيانة النظم البرمجية الحديثة.

الكلمات المفتاحية: مراجعة الكود بالذكاء الاصطناعي، المراقبة، الإصلاح التلقائي، الوكلاء الذكيون، ضمان جودة البرمجيات.

Abstract

In recent years, AI-assisted software development has been widely adopted, increasing code production and accelerating delivery cycles. While this improves developer productivity, it also introduces new challenges in code review, quality assurance, and runtime reliability. Existing solutions typically address isolated stages of the software lifecycle, lacking a continuous connection between pre-merge analysis and post-deployment incident response. This fragmentation motivates the need for a unified platform that integrates code review and production observability within a single workflow. This thesis presents Falcon, an AI-driven observability platform designed to bridge this gap by unifying pre-merge and post-deployment processes. The system integrates a codebase graph engine for static dependency analysis using ts-morph, a sandboxed execution environment for secure code exploration, GitHub integration, and agent-based reasoning for structured code review and automated fix generation. It enables cross-file defect detection during pull request analysis, root-

cause investigation of runtime failures through Sentry integration, and automated patch generation with type-checking validation. A case study conducted on a production TypeScript monorepo demonstrates the feasibility of unifying code review and incident remediation within a single AI-assisted workflow. By combining repository analysis, runtime observability, and agent-based reasoning, Falcon supports continuous software quality assurance and contributes to improving the reliability and maintainability of modern software systems.

Keywords: AI-assisted code review, observability, automated remediation, agentic AI, software quality assurance.

Dedication

Dedication by BENNIA Souleymane Karim

To my father, Bennia Abdelhakim Fares,
To my mother, Yahia Aissa Samira,
To my elder brother, Abdellatif Saber, and his wife Berrahia Safia,
and their children Lina Chahd, and Rayan Mohamed,
To my brother, Mohamed Elmahdi,
To my younger brother, Fouad,
To my best friends, Barkat Abdelmomin, Benyounes Mehdi,
and Zaoui Islam Mohamed Lamine,
And to the rest of my family and everyone who supported me
in any way, this work is yours as much as it is mine.

Dedication by Benyounes Mehdi

To my father, Benyounes Noureddine (رحمه الله), whose love, guidance, and memory
continue to accompany me every day,
To my mother, Khoudour Fatima Zahra,
To my brother, Mohammed Anis,
To my sister, Malika,
To my best friends, Barkat Abdelmomin, BENNIA Souleymane Karim,
and Zaoui Islam Mohamed Lamine,
To my whole family, my friends, and colleagues who stood by me,
And to everyone who believed in me and supported me
throughout this journey, this work is dedicated to you.

Acknowledgements

We are deeply indebted to our thesis supervisor, Dr. Debbi Hichem, whose steadfast support, guidance, and inspiration made this project a success. In a very special way, we thank him for his continuous encouragement and patience throughout this challenging study.

We also extend our sincere gratitude to the university and its administration for the support and guidance provided during our research. The resources and environment they offered were instrumental in bringing this work to completion.

Special thanks go to our friends and families, who endured the hectic moments and stood by us through every stage of this journey.

Last but not least, we thank every group member for their solidarity, commitment, and the shared effort that turned this thesis into a reality. Our collaboration taught us more than any textbook could.

Contents

Abstract **i**

Introduction **1**

1 Preliminary Study: Software's Evolution **3**
1.1 Introduction . . . 3
1.2 Evolution of Software Development . . . 3
1.2.1 Early Software Development . . . 3
1.2.2 Collaborative Software Development . . . 3
1.2.3 Version Control Systems . . . 4
1.2.4 Collaborative Platforms and Pull Requests . . . 4
1.2.5 Continuous Integration and Modern Development Practices . . . 5
1.3 AI-Assisted Software Development . . . 6
1.3.1 Integration of AI into Development Workflows . . . 6
1.3.2 AI Code Generation and Productivity Growth . . . 7
1.3.3 Rise in Code Volume and Pull Requests . . . 7
1.3.4 Benefits and Emerging Challenges . . . 8
1.3.5 Transition Toward Intelligent Development Ecosystems . . . 9
1.4 Industry Surveys and Empirical Trends in Modern Software Development 9
1.4.1 Growth of Collaborative Development . . . 10
1.4.2 Adoption of AI-Assisted Development Tools . . . 10
1.4.3 Quality, Reliability, and Security Concerns . . . 12
1.5 Conclusion . . . 12

2 Foundations and Related Concepts **13**
2.1 Introduction . . . 13
2.2 Large Language Models . . . 13
2.2.1 Definition of Large Language Models . . . 13
2.2.2 Tokenization and Next-Token Prediction . . . 14
2.2.3 Pretraining and Fine-Tuning . . . 14
2.2.4 Context Window and Prompt-Based Inference . . . 16
2.3 Transformers . . . 17
2.3.1 Transformer Architecture Overview . . . 17
2.3.2 Self-Attention Mechanism . . . 18

2.3.3 Positional Encoding . . . 18
2.3.4 Multi-Head Attention . . . 18
2.3.5 Relevance to Code and Technical Text . . . 18
2.4 AI Agents . . . 19
2.4.1 Definition of AI Agents . . . 19
2.4.2 Goal Decomposition and Planning . . . 20
2.4.3 Tool Use and External Interaction . . . 21
2.4.4 Memory and Iterative Execution . . . 22
2.4.5 Human Oversight and Control . . . 23
2.5 Agentic AI . . . 24
2.5.1 Agentic Workflow Overview . . . 24
2.5.2 Perception and Context Acquisition . . . 24
2.5.3 Reasoning and Decision-Making . . . 24
2.5.4 Orchestration of Multiple Agents . . . 24
2.5.5 Learning, Reflection, and Operational Risks . . . 25
2.6 Conclusion . . . 25

3 Code Review and Real-World Applications 26
3.1 Introduction . . . 26
3.2 Code Review in Software Development . . . 26
3.2.1 Role of Code Review . . . 26
3.2.2 Pull Request-Based Review Workflow . . . 26
3.2.3 Tool Support and Operational Limits . . . 27
3.2.4 Operational Best Practices . . . 27
3.3 Real-World AI Code Review Applications . . . 28
3.3.1 Greptile . . . 28
3.3.1.1 Definition . . . 28
3.3.1.2 How It Works . . . 28
3.3.2 CodeRabbit . . . 29
3.3.2.1 Definition . . . 29
3.3.2.2 How It Works . . . 29
3.3.3 Vercel AI Agent . . . 30
3.3.3.1 Definition . . . 30
3.3.3.2 How It Works . . . 30
3.3.4 Cubic AI . . . 31
3.3.4.1 Definition . . . 31
3.3.4.2 How It Works . . . 31
3.3.5 RovoDev . . . 32

3.3.5.1 Definition . . . 32
3.3.5.2 How It Works . . . 32
3.4 Benefits and Limitations of AI-Assisted Code Review . . . 33
3.4.1 Benefits . . . 33
3.4.2 Limitations . . . 33
3.5 Comparative Analysis . . . 34
3.6 Conclusion . . . 34

4 Design, Implementation, and Evaluation 35
4.1 Introduction . . . 35
4.2 Problem Statement . . . 35
4.3 Proposed Solution Overview . . . 35
4.4 Design and Implementation . . . 37
4.4.1 Requirements and Design Goals . . . 37
4.4.1.1 Design Goals . . . 37
4.4.1.2 Functional Needs . . . 37
4.4.1.3 Non-functional Needs . . . 38
4.4.1.4 Use Cases . . . 39
4.4.2 Core Concepts . . . 40
4.4.3 Technology Stack . . . 40
4.4.4 Monorepo Package Overview . . . 41
4.4.5 System Architecture Overview . . . 43
4.4.6 Codebase Graph Engine . . . 45
4.4.6.1 Implementation Realization . . . 47
4.4.7 Sandbox Layer . . . 48
4.4.7.1 Implementation Realization . . . 50
4.4.8 Reviewer Agent . . . 51
4.4.8.1 Multi-Agent Architecture . . . 51
4.4.8.2 Two-Phase Prompting Strategy . . . 52
4.4.8.3 Specialized Sub-Agents and Roles . . . 52
4.4.8.4 Tooling Layer . . . 54
4.4.8.5 Diff Summarizer . . . 55
4.4.8.6 Output Schema and Severity Model . . . 55
4.4.8.7 Orchestrator . . . 56
4.4.8.8 Reliability and Guardrails . . . 56
4.4.8.9 Implementation Realization . . . 57
4.4.9 Mechanic Agent . . . 58
4.4.9.1 Implementation Realization . . . 60

4.4.10 GitHub Integration 60
4.4.10.1 Implementation Realization 60
4.4.11 Data Model Design 61
4.4.12 Web Application Design 64
4.5 Application Workflow 64
4.5.1 GitHub App Installation and Repository Connection 64
4.5.2 Code Review Workflow 66
4.5.3 Sentry Integration and Issue Sync Workflow 67
4.5.4 Mechanic Agent Workflow 68
4.6 Evaluation 71
4.6.1 Evaluation Paradigms for AI Code Review 71
4.6.2 Selected Approach and Rationale 72
4.6.3 Evaluation Objectives 73
4.6.4 Evaluation Setup 73
4.6.4.1 Pull Request Case Selection 73
4.6.4.2 Execution Environment and Agent Configuration . . 74
4.6.4.3 Prompting Protocol and Fairness Controls 74
4.6.5 Metrics 74
4.6.6 Results 75
4.6.7 Comparative Evaluation of Automated Code Review Systems . 76
4.6.8 Discussion 76
4.6.9 Limitations 77
4.7 Challenges and Limitations 77
4.7.1 Project Scope and Limitations 77
4.7.2 Architectural Limitations 78
4.7.3 Technical Challenges Encountered 78
4.8 Conclusion 79

Conclusion **80**

References **82**

A Appendix: List of Acronyms **87**

List of Tables

4.1 Monorepo package structure and roles 42
4.2 Comparative metrics on PR #99: proposed agent vs. Cubic 75
4.3 Operational cost metrics reported for the proposed agent 75

List of Figures

1.1 Evolution of modern software development practices 6
1.2 Impact of AI-assisted development on pull request volume 8
1.3 Number of new developers on GitHub over time 10
1.4 AI sentiment and usage among developers 11
1.5 AI integration in IDEs 11

2.1 The workflow of the RLHF algorithm [35] 15
2.2 Overview of a large language model pipeline 16
2.3 Basic AI agent execution loop 20
2.4 Agentic AI workflow and orchestration 25

3.1 Traditional pull request-based review workflow 27
3.2 Visualized Greptile repository graph construction output [42] 29
3.3 CodeRabbit workflow 30
3.4 RovoDev workflow 33

4.1 Use case diagram 39
4.2 Monorepo package dependency graph showing runtime dependencies between workspace packages 43
4.3 Overview of the proposed AI-powered code review system 44
4.4 System-wide data flow showing interactions between external services, core components, agents, and persistence layer 44
4.5 Two-pass graph construction: Pass 1 registers nodes and extracts declarations; Pass 2 resolves cross-references 46
4.6 Sandbox lifecycle state machine with six states and transition conditions 49
4.7 Reviewer multi-agent pipeline: context assembly, parallel specialist execution, validation, orchestration, and fallback 52
4.8 End-to-end PR review sequence: from GitHub webhook through sandbox provisioning, parallel specialist execution, orchestration, and result publication 57
4.9 Mechanic agent five-stage protocol with verification retry loop and Draft PR creation 59
4.10 Entity-relationship diagram showing the data model organized into shared scaffolding, review pipeline, and remediation pipeline 63
4.11 Initiate GitHub App installation from settings page 64

4.12 Redirect to GitHub App installation 65
4.13 GitHub App permissions 65
4.14 Repository access scope selection 65
4.15 Connected repositories visible in dashboard 66
4.16 Pull request event reception 66
4.17 Streaming findings as GitHub annotations and comments 67
4.18 Connect Sentry from settings page 67
4.19 Sentry authorization and connection 68
4.20 Sentry issues fetched and listed in platform 68
4.21 User-triggered Fix action 69
4.22 Branch creation and publication to remote repository 70
4.23 Created pull request from mechanic agent 71

Introduction

AI-assisted coding tools have increased the speed and volume of software production, but code review and quality assurance processes have not scaled at the same rate. Teams face denser review queues and larger diffs, conditions under which cross-file defects are more likely missed (i.e., a developer changes a function signature in one file but forgets its call sites, and the reviewer does not trace every dependency). At the same time, runtime failures in production require manual debugging that does not feed back into the code review pipeline, so the same error class can recur across different pull requests. Existing AI-powered tools cover individual phases in isolation: Greptile (repository-structure analysis at PR time), CodeRabbit (iterative diff review within PRs), Cursor (AI-native code authoring with embedded self-review), and Vercel (deployment telemetry linked to code), but none unifies static code analysis with runtime observability in a closed remediation loop (i.e., an error surfaces in production, gets traced to its root cause, triggers a code fix, validates it, and creates a reviewable pull request, all within the same system). This leads to our research question: can a single platform connect pre-merge code review with post-deployment incident remediation without fine-tuning on proprietary data? We present Falcon, a platform that answers this question through three components. First, a Codebase Graph Engine that builds static dependency graphs from TypeScript source files using ts-morph (i.e., a library exposing the TypeScript compiler API for symbol resolution). Second, a Reviewer Agent that uses a two-phase prompt strategy: Cold Read, where the model inspects the full diff without tools, then Explore Mode, where it queries the codebase for supporting evidence. Third, a Mechanic Agent that traces errors from Sentry (a runtime error tracking platform) through the dependency graph, applies hypothesis-driven patches, and validates through test execution, type checking, and linting before generating draft pull requests. Our scope is limited to TypeScript monorepos using GitHub and Sentry. Our objectives are to detect defects, security risks, and maintainability issues at PR time; to automate root-cause analysis and fix generation for runtime failures; and to evaluate both pipelines on a production repository. The main contributions are:

- a unified platform that improves review quality by detecting defects humans miss while reducing reviewer workload (i.e., automating routine checks so reviewers can focus on architectural decisions);
- a dual-agent design that separates detection from automated fix generation,

each with validated correctness guarantees (i.e., the Reviewer flags issues, the Mechanic fixes them, both pass type-checking and linting before reaching a human);

- a cross-file analysis approach based on static dependency graphs rather than prompt-based analysis of isolated diffs (e.g., Atlassian's RovoDev augments the prompt with PR metadata and Jira context but still relies on the LLM to reason about structural dependencies from flat text [1]; Falcon's graph explicitly models caller-callee relationships and cross-package boundaries that the model cannot infer from the diff alone);

The remainder of this thesis is organized as follows:

- Chapter 1 describes the evolution of software development and how AI has changed review workflows. It also sets up the practical problems that motivate the rest of the thesis.
- Chapter 2 covers the background needed to understand the system, including LLMs, Transformers, and agentic AI. It explains the concepts that Falcon builds on.
- Chapter 3 examines current code review practices and industrial AI-assisted review solutions, analyzes their strengths and limitations, and presents the proposed solution developed in this work.
- Chapter 4 presents Falcon's design, implementation, and evaluation. It explains how the platform works in practice and what the experiments show.

Through this project, we combine recent advances in artificial intelligence and software engineering to develop a practical platform that assists developers in maintaining software quality, improving review efficiency, and responding more effectively to production failures.

Chapter 1 Preliminary Study: Software's Evolution

1.1. Introduction

AI-assisted code generation has accelerated software production, but review and debugging processes remain manual and disconnected from each other. Development teams now combine automated pipelines, version control platforms, and AI tools to increase productivity, while facing growing challenges in maintaining software quality and reliability. This chapter examines how collaborative and AI-augmented development practices have widened the gap between production speed and quality assurance, and how this gap motivates the design of Falcon.

1.2. Evolution of Software Development

1.2.1. Early Software Development

Early software was built by individual programmers or small teams working on isolated systems [2]. Code was managed through manual file backups and exchanged directly between developers [3], yet these approaches became unreliable as systems grew larger and more complex [4].

As projects expanded, teams encountered recurring problems: overwritten files, inconsistent versions, integration conflicts, and difficulty tracking who changed what [5]. Without structured collaboration, long-term maintenance and evolution became hard to sustain [6].

1.2.2. Collaborative Software Development

As the software industry grew, development shifted to distributed teams working on shared codebases simultaneously [7]. Synchronization, communication, dependency management, and integration of concurrent changes became critical coordination challenges [8].

Development methodologies evolved to address these challenges through iterative, collaborative models [9]. Agile practices introduced short cycles, continuous feedback,

incremental delivery, and frequent integration [10]. Software engineering shifted from isolated implementation toward team-oriented workflows centered on rapid iteration and collaboration [11], and modern practices such as peer review, issue tracking, sprint planning, and shared repository management improved coordination and code quality [12].

1.2.3. Version Control Systems

Version control systems solved the coordination problem by tracking code changes and preserving project history systematically. Distributed systems like Git gained widespread adoption because they let developers work independently while keeping a complete repository copy locally [2].

Git introduced lightweight branching and merging strategies that enabled parallel development without affecting the main codebase. Developers could isolate experiments, maintain feature-specific branches, and merge changes through controlled workflows.

Version control systems provide several advantages:

- Tracking source code evolution over time.
- Restoring previous project states when necessary.
- Supporting parallel and distributed development.
- Reducing integration conflicts between contributors.
- Improving collaboration and traceability.

These capabilities made large-scale collaboration practical and established the foundation for modern review workflows.

1.2.4. Collaborative Platforms and Pull Requests

Platforms like GitHub, GitLab, and Bitbucket extended version control systems by integrating repository hosting, issue tracking, project management, automation, and review mechanisms into centralized environments [13].

The Pull Request model became the most influential workflow these platforms introduced [14]. Developers submit proposed modifications through isolated branches

that are reviewed before merging into the main codebase [15]. Pull Requests structure the discussion of implementation decisions, defect identification, and compliance with project standards [16].

Pull Requests play a critical role in quality assurance because they enable teams to:

- Review code before integration [17].
- Detect bugs and potential vulnerabilities [18].
- Verify adherence to coding standards and architectural guidelines [16].
- Discuss implementation strategies and design choices [17].
- Maintain traceability of development decisions [18].

As collaborative development matured, Pull Request-based review became a central practice in professional software engineering [14].

1.2.5. Continuous Integration and Modern Development Practices

As systems grew more complex, organizations adopted automation to improve reliability, scalability, and delivery speed. Continuous Integration introduced automated pipelines that validate source code through compilation, testing, and analysis whenever changes are submitted [12].

Continuous Deployment extended this process by automating delivery and release workflows, reducing manual effort during deployment. These practices shifted software engineering to a highly automated lifecycle where integration and validation happen continuously rather than at isolated release stages.

Modern CI/CD pipelines commonly integrate:

- Automated unit and integration testing.
- Static code analysis.
- Security scanning tools.
- Build and deployment automation.
- Quality assurance checks.

These practices provide several benefits:

- Faster development and delivery cycles.
- Early detection of integration issues.
- Improved software stability and reliability.
- Reduced manual operational overhead.
- Continuous validation of software quality.

Software development thus evolved into a collaborative, automated ecosystem that later enabled the integration of AI-assisted workflows.

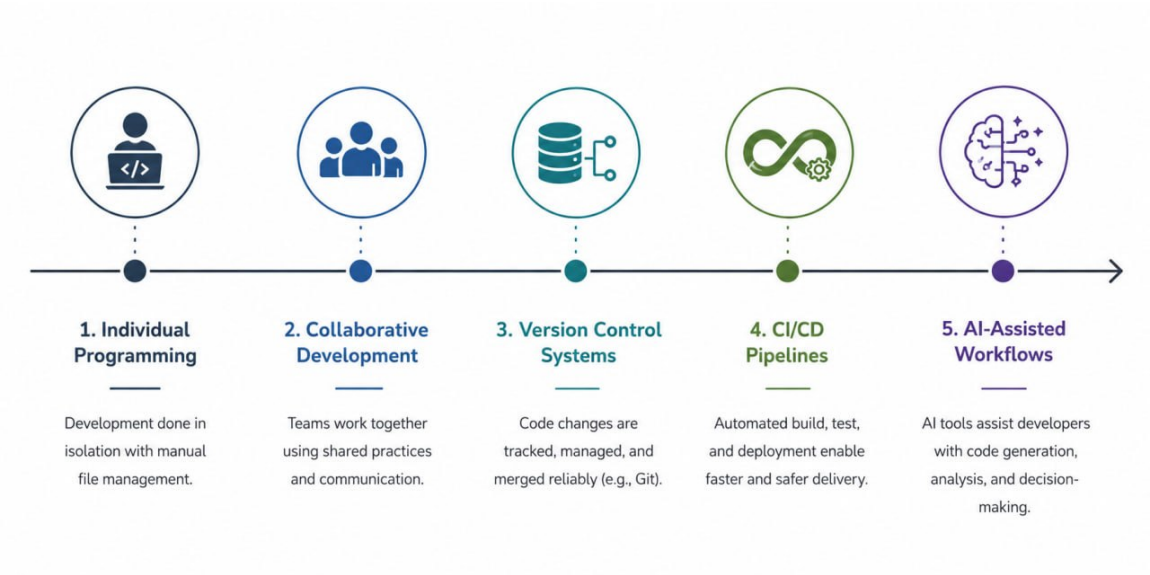


Figure 1.1: Evolution of modern software development practices

1.3. AI-Assisted Software Development

1.3.1. Integration of AI into Development Workflows

AI-assisted tools now integrate directly into IDEs, repository platforms, and collaboration systems. Rather than replacing developers, these systems act as interactive assistants that provide contextual recommendations during implementation [19].

Modern AI-assisted workflows commonly include:

- Context-aware code completion.
- Intelligent documentation assistance.
- Real-time syntax and semantic suggestions.
- Automated refactoring recommendations.
- AI-supported debugging assistance.

This integration reduces repetitive manual tasks and accelerates development, letting developers focus more on architecture, design, and business logic.

1.3.2. AI Code Generation and Productivity Growth

The most visible impact of AI in software engineering is faster code production [19]. LLMs generate working code from natural language instructions, partial implementations, or project context [20].

In practice, AI tools are used to:

- Generate project scaffolding [21].
- Produce boilerplate code [22].
- Suggest implementation logic [23].
- Complete partially written functions [19].
- Assist debugging and refactoring tasks [21].

These capabilities shorten development time and implementation cycles [23]. However, faster code generation also changes the dynamics of review and validation [24].

1.3.3. Rise in Code Volume and Pull Requests

AI-assisted development increases the volume of code produced within the same timeframe. Developers generate larger implementations faster, which directly impacts Pull Request workflows [19].

This productivity increase creates operational challenges:

- More frequent Pull Request submissions.
- Larger and more complex code changes.
- Shorter expected review cycles.
- Increased reviewer cognitive load.
- Greater pressure on quality assurance processes.

As development throughput increases, review and validation become bottlenecks in the delivery pipeline.

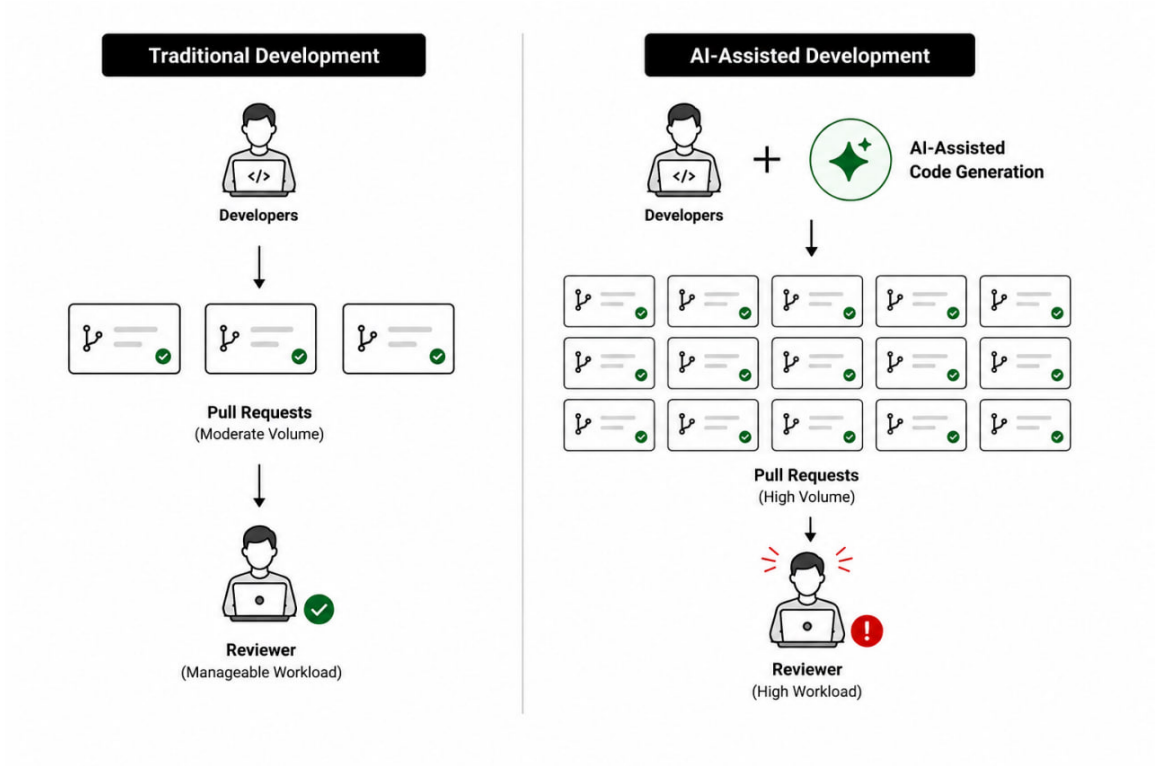


Figure 1.2: Impact of AI-assisted development on pull request volume

1.3.4. Benefits and Emerging Challenges

AI-assisted development improves productivity, accessibility, and implementation speed, but introduces new technical and organizational challenges. Generated code may be syntactically correct while containing semantic errors, hidden vulnerabilities, performance issues, or maintainability problems.

Major challenges introduced by AI-assisted development include:

- Inconsistent code quality across generated outputs.
- Increased risk of unnoticed bugs and vulnerabilities.
- Reduced developer understanding of generated implementations.
- Greater dependency on automated suggestions.
- Higher workload on code review processes.

These challenges show that faster code generation must be matched by equally effective validation and quality assurance mechanisms.

1.3.5. Transition Toward Intelligent Development Ecosystems

Software engineering is evolving toward intelligent development ecosystems where generation, testing, deployment, monitoring, and review are increasingly augmented by AI [25]. Modern workflows combine automation with human oversight to improve scalability and reliability [26].

Despite automation advances, human supervision remains essential for evaluating correctness, maintainability, architecture quality, security implications, and business requirements [27]. This gap between automated generation and reliable validation motivates the need for more advanced AI-powered review systems capable of assisting developers throughout the software lifecycle [28].

1.4. Industry Surveys and Empirical Trends in Modern Software Development

The evolution of software engineering is reflected in industry surveys and empirical studies published by major development platforms and research organizations. These reports provide quantitative evidence on the growth of collaborative development, automation adoption, and the increasing integration of AI tools into software engineering workflows.

1.4.1. Growth of Collaborative Development

Modern software engineering has become increasingly collaborative and platform-driven. GitHub reports indicate continued growth in the number of developers participating in distributed version control workflows and pull request-based collaboration models.

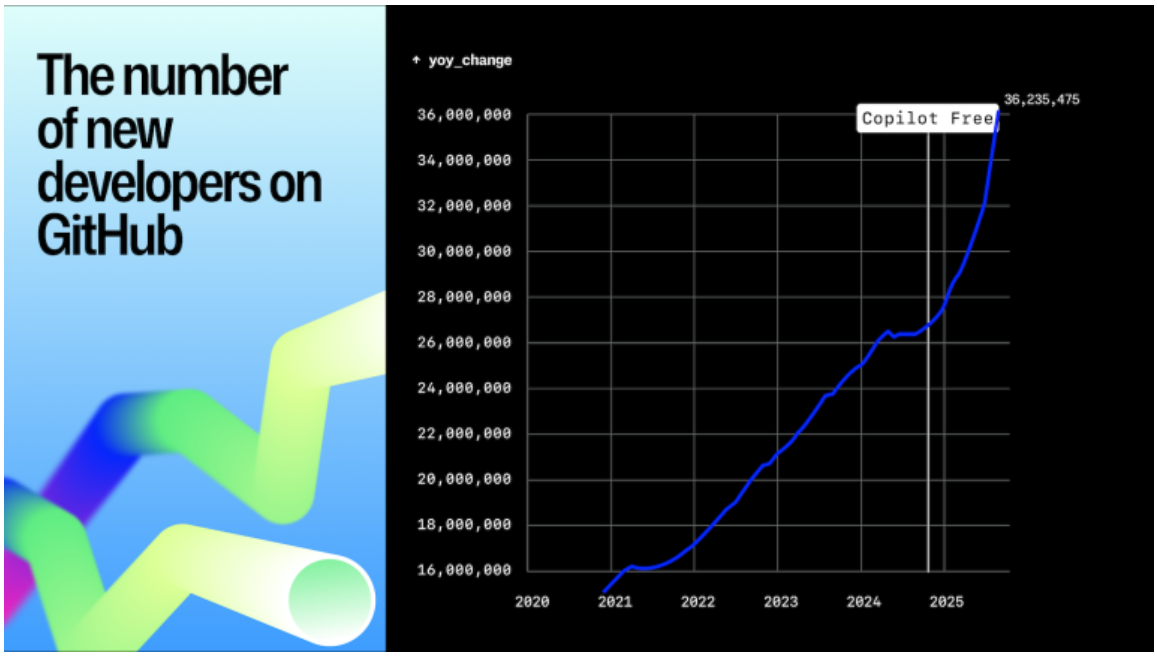


Figure 1.3: Number of new developers on GitHub over time [29]

Pull request activity reached record levels in 2025: pull requests merged averaged 43.2 million per month, up from 35 million in 2024, while pull requests created increased by 20.4% to 47.5 million from 39.5 million. This growth reinforces how pull request workflows have become a central coordination mechanism for collaborative software development [14].

CI/CD survey results show that 62% of respondents use GitHub Actions for personal projects and 41% use it in organizations, while 32% of organizations use two CI/CD tools and 9% use at least three. The same survey also found that 73% of respondents do not use AI in their CI/CD workflows, showing that adoption is still measured despite growing automation interest [30].

1.4.2. Adoption of AI-Assisted Development Tools

Recent surveys show rapid adoption of AI-assisted development tools among software engineers. Reports from Stack Overflow, GitHub, and JetBrains indicate that devel-

opers increasingly use AI for code completion, documentation generation, debugging assistance, refactoring, and rapid prototyping.

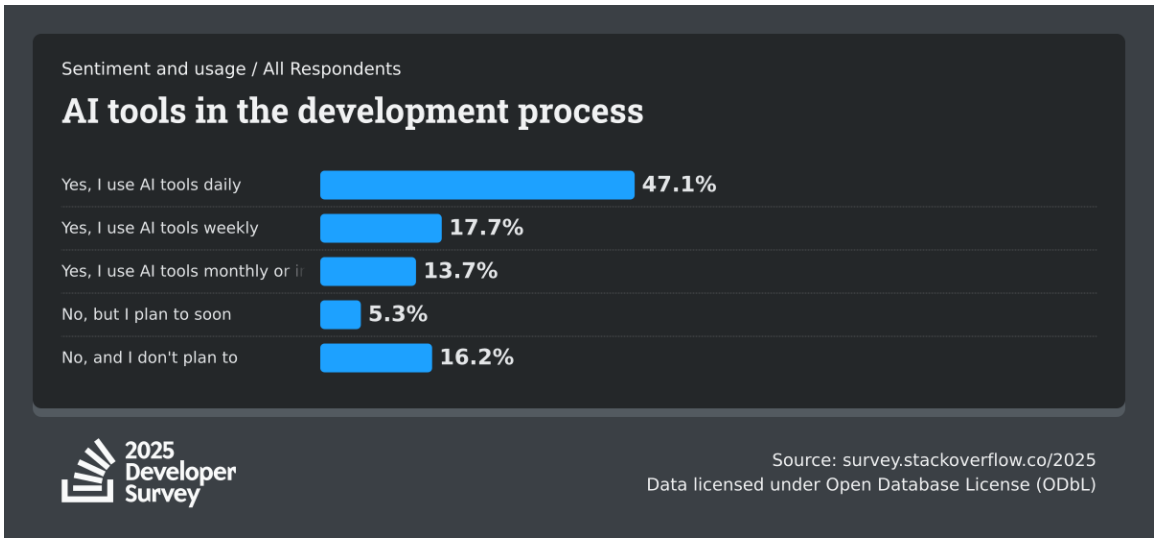


Figure 1.4: AI sentiment and usage among developers [31]

AI-assisted systems are now integrated directly into IDEs, allowing developers to receive contextual suggestions during implementation.

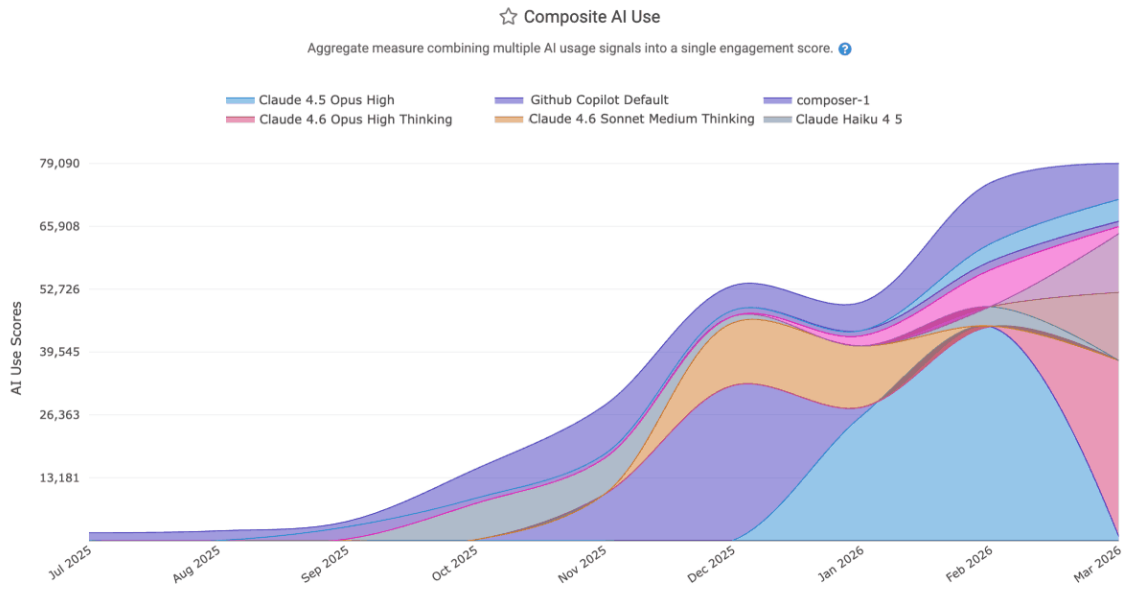


Figure 1.5: AI integration in IDEs [32]

Commonly used tools include GitHub Copilot, Cursor, and ChatGPT-based assistants integrated into development workflows.

Empirical studies on AI coding assistants report measurable productivity improvements, especially for boilerplate generation, function completion, and rapid experimentation.

1.4.3. Quality, Reliability, and Security Concerns

Despite productivity gains, multiple empirical studies highlight concerns about the quality and reliability of AI-generated code. Reported issues include logical defects, insecure patterns, incorrect API usage, hallucinated dependencies, and maintainability problems.

Recent CodeRabbit analysis reports that AI-generated code has 1.7x more issues and bugs than human-written code, with logic and correctness problems among the most common and security issues landing 1.5x to 2x higher. The same report also found that AI-generated code is 2.25x more likely to create algorithmic and business logic errors and 2.29x more likely to introduce incorrect concurrency-control issues [24], [33].

The quality gap extends beyond pre-merge review. When defects reach production, debugging remains entirely manual: a developer must receive the alert, reproduce the failure, trace the root cause, and apply a fix. This process is disconnected from the review pipeline, so the same error type can appear in multiple pull requests without a structured feedback loop connecting runtime behavior back to code review. The growing volume of AI-generated code that passes review but carries subtle defects amplifies this disconnection.

These conditions increase the importance of automated review pipelines, static analysis tools, and structured human oversight. They motivate scalable AI-assisted review mechanisms that combine automation with human judgment.

1.5. Conclusion

The evidence in this chapter establishes that collaborative development and AI-assisted coding have created a widening gap between production speed and quality assurance. Code review remains manual and does not scale with the volume of code produced. Runtime failures require disconnected debugging with no structured connection to the development pipeline. These limitations define the reliability gap that Falcon is designed to address.

Chapter 2 Foundations and Related Concepts

2.1. Introduction

This chapter introduces the technical foundations that support the thesis context by explaining the main ideas behind Large Language Models, Transformer architectures, AI agents, and agentic AI, while focusing on the concepts needed to understand how modern AI systems process language, model dependencies, use tools, and support software workflows.

2.2. Large Language Models

2.2.1. Definition of Large Language Models

Large Language Models (LLMs) are deep neural models trained on very large text and code corpora to model token sequences and generate coherent natural language or source-code outputs. Their behavior is statistical rather than rule-based: instead of following hard-coded logic, the model learns distributed representations of syntax, semantics, style, and domain-specific patterns from data [34].

At a high level, an autoregressive LLM estimates the probability of a sequence by factorizing it token by token:

$$P(x_{1:T}) = \prod_{t=1}^{T} P(x_t \mid x_{<t}) \tag{2.1}$$

In this formulation:

- $P(x_{1:T})$ denotes the joint probability of the entire token sequence from position 1 to T.
- T is the total length of the sequence (number of tokens).
- x_t represents the token at position t in the sequence.

- $x_{<t}$ denotes the prefix, i.e., all tokens preceding position t.
- $P(x_t \mid x_{<t})$ is the conditional probability of token x_t given the prefix $x_{<t}$.
- $\prod_{t=1}^{T}$ indicates that the joint probability is obtained by multiplying the successive conditional probabilities for every token in the sequence.

This factorization models sequence generation as a conditional process in which each token depends on the prefix generated so far. The learned parameters approximate these conditional distributions from large corpora, capturing recurrent statistical regularities while remaining limited to patterns observed during training.

2.2.2. Tokenization and Next-Token Prediction

Before inference, input text is tokenized into smaller units such as words, subwords, or characters. The tokens are mapped to embeddings, which are then transformed through the network into contextual representations. This token-based processing is important because it allows the model to handle open vocabulary input, including technical identifiers, rare domain terms, and source-code fragments [34].

Next-token prediction is therefore a probabilistic selection over a high-dimensional vocabulary rather than a deterministic continuation rule. Output quality depends on the extent to which the model representation preserves syntactic structure, semantic coherence, and task-specific constraints within the available context.

2.2.3. Pretraining and Fine-Tuning

LLM development is typically a two-stage process. Pretraining exposes the model to broad linguistic and structural patterns using massive unlabeled datasets. Fine-tuning then adapts the pretrained model to narrower behaviors such as summarization, instruction following, classification, or code-related tasks [34].

Common adaptation strategies include supervised fine-tuning, reinforcement learning from human feedback, and instruction tuning. Supervised fine-tuning narrows model behavior toward labeled examples and can reduce generality outside the target distribution. Reinforcement learning from human feedback shifts outputs toward preferred responses, although preference does not always align with factual correctness. Instruction tuning improves request following, but it can also obscure uncertainty in the underlying prediction.

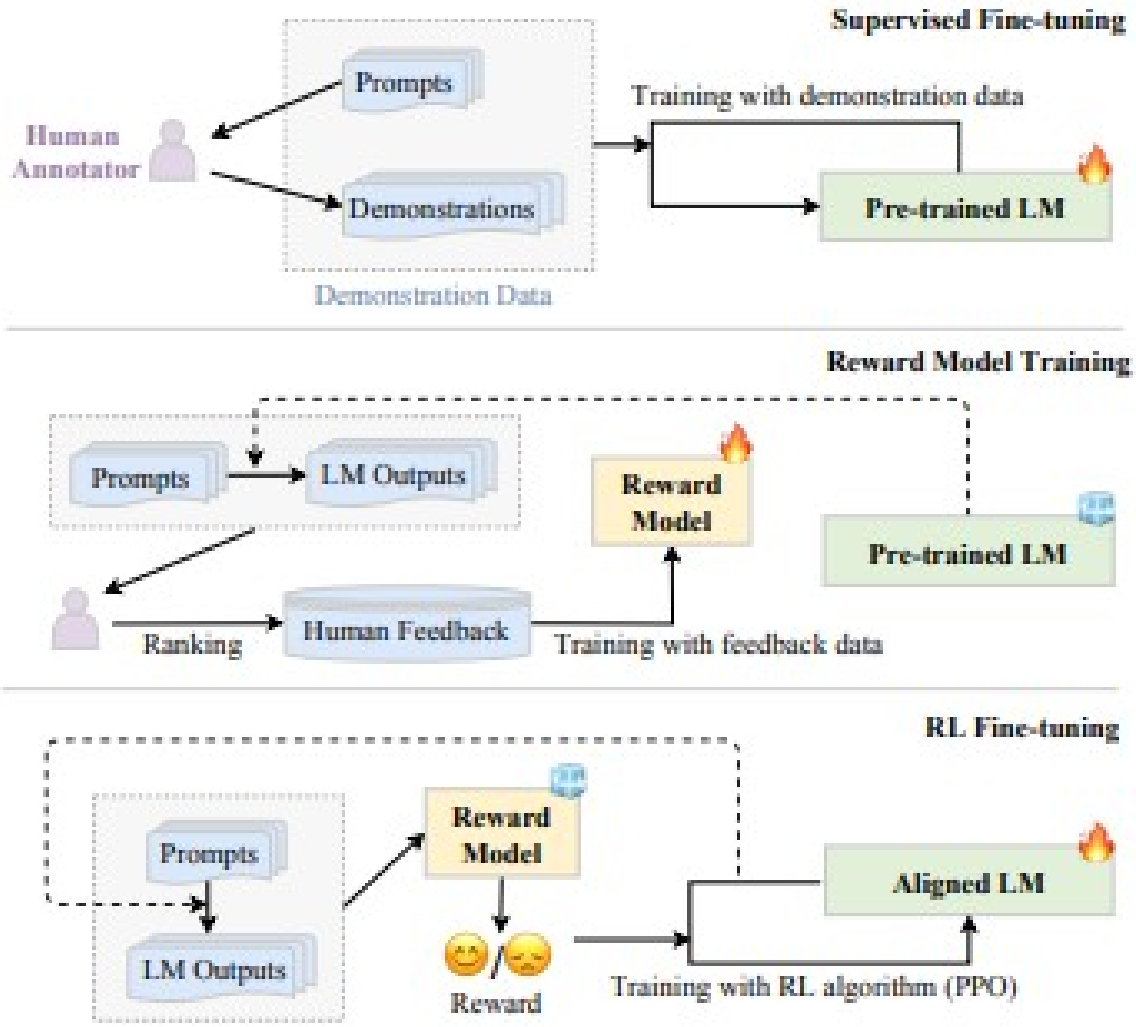


Figure 2.1: The workflow of the RLHF algorithm [35]

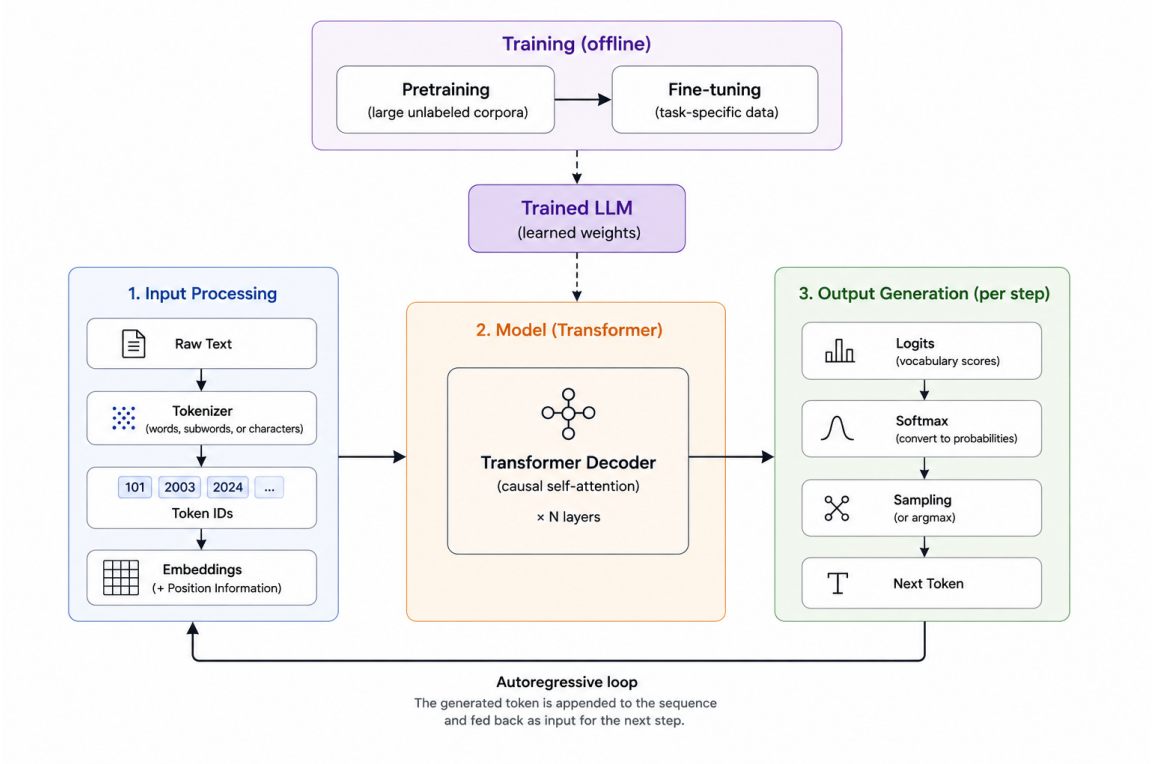


Figure 2.2: Overview of a large language model pipeline

2.2.4. Context Window and Prompt-Based Inference

During inference, the prompt provides the immediate task description and the context window defines how much prior information the model can condition on at once. Longer context windows are particularly useful for technical documents, code assistance, and multi-turn interaction because they allow the model to preserve more surrounding information [34].

LLMs remain probabilistic systems and therefore can generate hallucinated outputs, reproduce training-data bias, or produce code and explanations that are structurally plausible but semantically incorrect. Their behavior is bounded by prompt quality, context-window capacity, and the coverage of the training distribution, which together define the space of outputs the model can represent reliably.

In software tasks, LLMs can support code explanation, summarization, documentation, and repair suggestions, but they do not guarantee correctness. Generated outputs should therefore be treated as provisional artifacts and validated through deterministic checks, human review, or external tooling.

2.3. Transformers

2.3.1. Transformer Architecture Overview

The Transformer architecture is the neural architecture underlying most modern LLMs. It is a sequential-data architecture that relies on attention rather than recurrence. This design choice allows the model to process sequences in parallel while still modeling relationships between distant elements in the input [36].

The central computation in a Transformer layer is scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.2}$$

In this formulation:

- Q (Queries) is a matrix of learned projections derived from the input embeddings, representing what each token is ``looking for.''
- K (Keys) is a corresponding matrix of learned projections representing what each token ``offers.''
- V (Values) is a third matrix of learned projections containing the actual content to be aggregated.
- QK^T computes the dot product between every query and every key, yielding a raw relevance score (attention score) for each pair of tokens.
- d_k is the dimension of the key vectors, used to scale the scores.
- $\sqrt{d_k}$ is the scaling factor that stabilizes gradients by preventing the dot products from growing too large as dimensionality increases.
- softmax$(\cdot)$ normalizes the scaled scores into a probability distribution so that the scores sum to one, turning them into attention weights.
- The product with V produces a weighted sum of values, yielding a context-aware representation for each token based on the most relevant parts of the sequence.

This operation measures pairwise relevance and produces a context-sensitive representation that captures token dependencies within the sequence.

2.3.2. Self-Attention Mechanism

Self-attention is the mechanism that lets each token attend to every other token in the same sequence. The output representation for each token therefore depends on the full surrounding context rather than only on the most recent hidden state. This is especially valuable for technical text and source code because the meaning of a token often depends on distant references such as import statements, function calls, variable declarations, comments, and configuration values [36].

2.3.3. Positional Encoding

Positional encoding is required because self-attention alone does not encode order. By injecting positional information into embeddings, the Transformer preserves sequence structure and can distinguish between different token arrangements even when the same tokens appear in a different order. In source code, this is important because ordering affects control flow, argument binding, and nesting structure [36].

2.3.4. Multi-Head Attention

Multi-head attention extends self-attention by computing several attention projections in parallel. Each head can specialize in a different relation, such as local syntax, long-range dependency, or identifier matching. After the separate heads produce their outputs, the results are concatenated and passed through feed-forward layers, residual connections, and normalization layers [36].

The mechanism removes the strict locality assumptions of recurrent architectures, so dependencies can be represented regardless of positional distance. The trade-off is quadratic computational and memory complexity with respect to sequence length, which becomes a limiting factor for long-context processing.

2.3.5. Relevance to Code and Technical Text

For code and technical text, the architecture is effective because it preserves long-range dependencies between identifiers, comments, function calls, configuration values, and surrounding documentation. The same mechanism also supports summarization, code completion, translation, and semantic matching in large technical contexts [36].

2.4. AI Agents

2.4.1. Definition of AI Agents

An AI agent is a system that combines a language model with an execution policy, a goal specification, and access to external tools. Rather than producing a single isolated response, the agent operates as a control loop that alternates between inference, action selection, observation, and state update. This structure extends the model beyond static text generation and enables it to coordinate multi-step tasks under explicit constraints [37].

An agentic stack typically includes a base model, a goal specification, a set of tools, a memory or state store, and an execution loop. The model interprets the input, the controller selects an action, the tool executes that action, and the resulting observation is fed back into the next decision step. The quality of the system depends on how effectively these components preserve task context while preventing unsupported assumptions from accumulating.

In practice, the model does not reason in isolation. Its decisions are bounded by tool availability, context length, and the policy used to rank candidate actions. This means that agent behavior is shaped not only by model capability but also by the surrounding orchestration layer, which determines when the agent should act, pause, or request additional information.

The main components are:

- **Base model:** generates candidate reasoning steps and action proposals from the current context.
- **Goal specification:** defines the target outcome and constrains the search space of possible actions.
- **Tools:** provide access to external state, structured data, and deterministic operations that the model cannot infer reliably from parameters alone.
- **Memory or state store:** preserves verified results, intermediate observations, and unresolved hypotheses across iterations.
- **Execution loop:** evaluates the next action, performs the tool call, and feeds the observation back into the agent state.

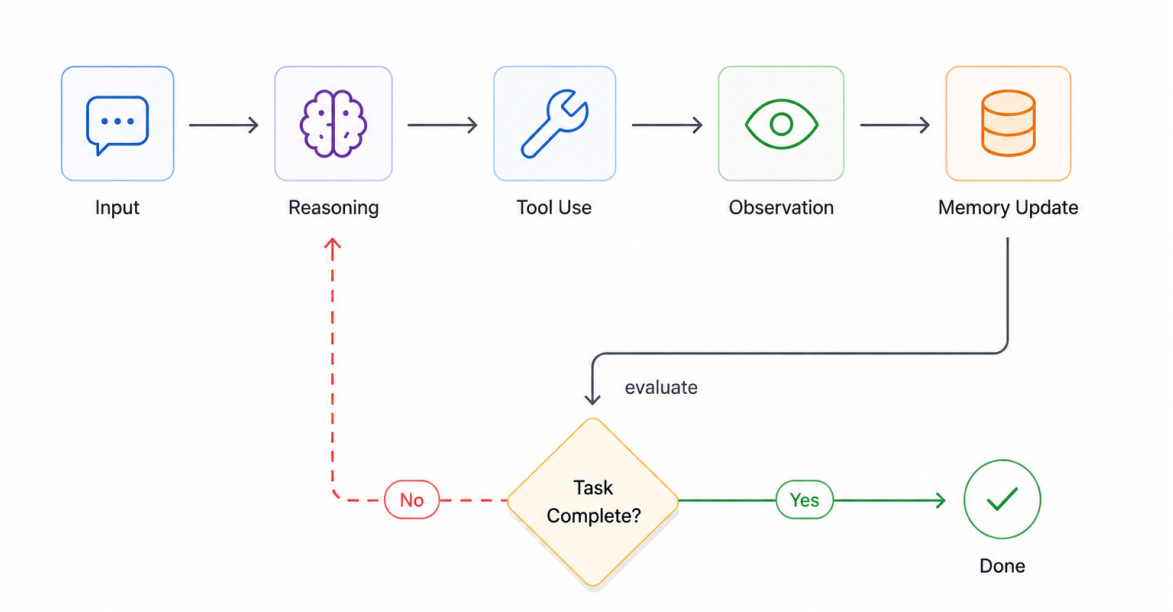


Figure 2.3: Basic AI agent execution loop

2.4.2. Goal Decomposition and Planning

Goal decomposition transforms a high-level objective into a sequence of subtasks that the agent can resolve iteratively. Human input defines the target, the applicable rules, and the available tools, after which the agent constructs a plan that assigns dependencies and ordering to the required actions. For simple objectives, this may be implemented as a short reflexive loop; for more complex objectives, the agent must generate a structured plan before execution [37].

The decomposition step is sensitive to prompt formulation and to the quality of the internal state representation. If the agent omits a dependency, selects an inefficient subtask order, or overcommits to an early hypothesis, the resulting plan can propagate the error through later steps and reduce the usefulness of the final output.

Planning also introduces a trade-off between breadth and precision. A shallow plan reduces token usage and latency, but it risks missing hidden dependencies. A deeper plan can improve coverage, yet it increases computational cost and creates more opportunities for divergence between the planned steps and the actual environment state.

Planning usually follows these stages:

1. Identify the final objective and the constraints that apply to it.
2. Decompose the objective into subtasks with explicit dependencies.

3. Order the subtasks so that prerequisite information is obtained before downstream actions.

4. Execute one subtask at a time and revise the plan when observations contradict the initial assumption.

2.4.3. Tool Use and External Interaction

Tool use separates reasoning from execution by delegating deterministic or up-to-date operations to external systems. Agents typically call search engines, databases, APIs, repositories, calculators, or specialized peer agents when internal parameters do not contain enough information to complete the task. After each tool call, the observation is normalized into the agent state so that subsequent reasoning can incorporate the new evidence [37].

This mechanism makes the agent more general-purpose than a standalone language model, but it also introduces dependency on tool quality, latency, and availability. If a tool returns incomplete, stale, or inconsistent data, the agent may still produce a plausible but incorrect conclusion because it assumes the observation is reliable.

Technically, tool calling requires the agent to convert an abstract intent into a machine-readable action schema, execute the call, and then map the response back into natural-language context. That translation layer is often the main source of failure, because mismatched parameters, ambiguous tool descriptions, or weak error handling can cause the agent to call the wrong function or misinterpret the result.

The main tool interactions are:

- **Retrieval tools:** search repositories, documentation, or web sources for missing facts.
- **Analytical tools:** parse, compare, summarize, or calculate results that require deterministic processing.
- **Action tools:** update files, trigger APIs, or execute operations in an external environment.
- **Specialized agents:** delegate subtasks to another agent with narrower expertise.

2.4.4. Memory and Iterative Execution

Memory allows the agent to retain intermediate results, prior actions, and feedback across steps. In practical systems, this memory may be implemented as a short-term context buffer, a task-specific scratchpad, or a longer-term store of past interactions. The main function of memory is to preserve what has already been established so that the agent can refine rather than restart its reasoning at every step [37].

Iterative execution uses this stored state to support reflection and self-correction. The agent can compare a new observation with prior assumptions, revise the next action, and avoid repeating unsuccessful tool calls. The limitation is that accumulated context can also become redundant, stale, or contradictory, which degrades decision quality unless the memory is filtered, summarized, or periodically reset.

This is why agent memory is usually treated as operational state rather than permanent knowledge. The system must distinguish between verified facts, tentative hypotheses, and obsolete observations; otherwise, earlier errors remain in the state and continue influencing later decisions.

In operational terms, memory is commonly used for:

- preserving validated intermediate results so that later steps do not recompute them;
- storing hypotheses that are still under evaluation;
- recording tool outputs for traceability and later review;
- summarizing long interactions so that the context window remains usable.

Several agent types are commonly distinguished, including simple reflex agents, model-based reflex agents, goal-based agents, utility-based agents, and learning agents. Simple reflex agents react only to the current perception, whereas model-based reflex agents maintain an internal state that compensates for partial observability. Goal-based agents search for action sequences that satisfy a target, utility-based agents rank alternatives by expected value, and learning agents update their behavior from experience. This classification captures increasing autonomy, but it also reflects increasing reliance on state estimation and policy selection.

The distinction is useful because it shows how complexity grows from fixed rule execution to adaptive decision-making. As agent capability increases, so does the

requirement for reliable memory, explicit objective functions, and feedback channels that can correct errors before they propagate.

2.4.5. Human Oversight and Control

Human oversight constrains agent behavior in scenarios where incorrect actions carry significant cost. In software engineering workflows, this includes validating generated code, approving external actions, and enforcing constraints on tool usage. Oversight becomes especially important when the agent can modify repositories, access live systems, or trigger downstream automation, because errors can propagate beyond the immediate response.

Supervision can be implemented through approval checkpoints, activity logs, interruption points, and identifier-based traceability. These controls do not remove autonomy, but they bound it by making the agent's actions observable, auditable, and interruptible when the system deviates from the intended goal.

In operational terms, oversight acts as a safety constraint on the agent loop. It limits high-impact actions, provides a manual fallback when the model is uncertain, and ensures that the final decision remains compatible with policy, security, and reliability requirements.

Common control mechanisms include:

- **Approval gates:** require human confirmation before external or irreversible actions.
- **Activity logs:** record tool calls, intermediate reasoning, and executed decisions.
- **Interruptibility:** allow operators to stop or redirect an agent that diverges from the goal.
- **Traceable identifiers:** associate actions with a specific agent instance, user, or workflow.

2.5. Agentic AI

2.5.1. Agentic Workflow Overview

Agentic AI extends an LLM from single-response generation to an iterative control loop that combines state tracking, tool selection, execution, and feedback. The agent receives a goal, decomposes it into substeps, evaluates available actions, and updates its internal state after each observation. Orchestration binds these steps into a bounded workflow so that the system can pursue a task without relying on one-shot generation alone [38].

2.5.2. Perception and Context Acquisition

Perception is the input stage that converts external signals into model-usable context. In practice, this may include prompt text, retrieved documents, API responses, file contents, or intermediate tool outputs. The agent filters this material into a working state representation, which limits the search space for subsequent decisions and reduces dependence on implicit assumptions in the base model [38].

2.5.3. Reasoning and Decision-Making

Reasoning and decision-making convert the current state into a next-action policy. ReAct-style loops [39] interleave internal reasoning with observation, allowing the agent to revise its plan after each tool result. Planning-oriented methods such as ReWOO separate plan construction from execution, which reduces redundant tool calls but increases sensitivity to initial plan quality. Both approaches depend on the quality of the state summary and on the agent's ability to detect when additional evidence is required [38].

2.5.4. Orchestration of Multiple Agents

Orchestration manages the interaction between agents, tools, and shared state. A controller assigns subtasks, routes intermediate outputs, and enforces execution order when one step depends on the result of another. In multi-agent settings, orchestration also resolves task ownership and conflict handling, which is necessary when different agents produce overlapping or contradictory outputs [38].

2.5.5. Learning, Reflection, and Operational Risks

Learning updates the agent's policy from feedback, whether that feedback comes from users, tool outputs, or supervisory signals. Reflection uses prior failures to adjust future actions, typically by revising memory, ranking alternative plans, or changing tool selection. This improves adaptation, but it also introduces risk when the stored context becomes stale, redundant, or inconsistent. Traceability, interruption points, and human supervision therefore remain necessary to constrain error propagation and keep the workflow auditable [38].

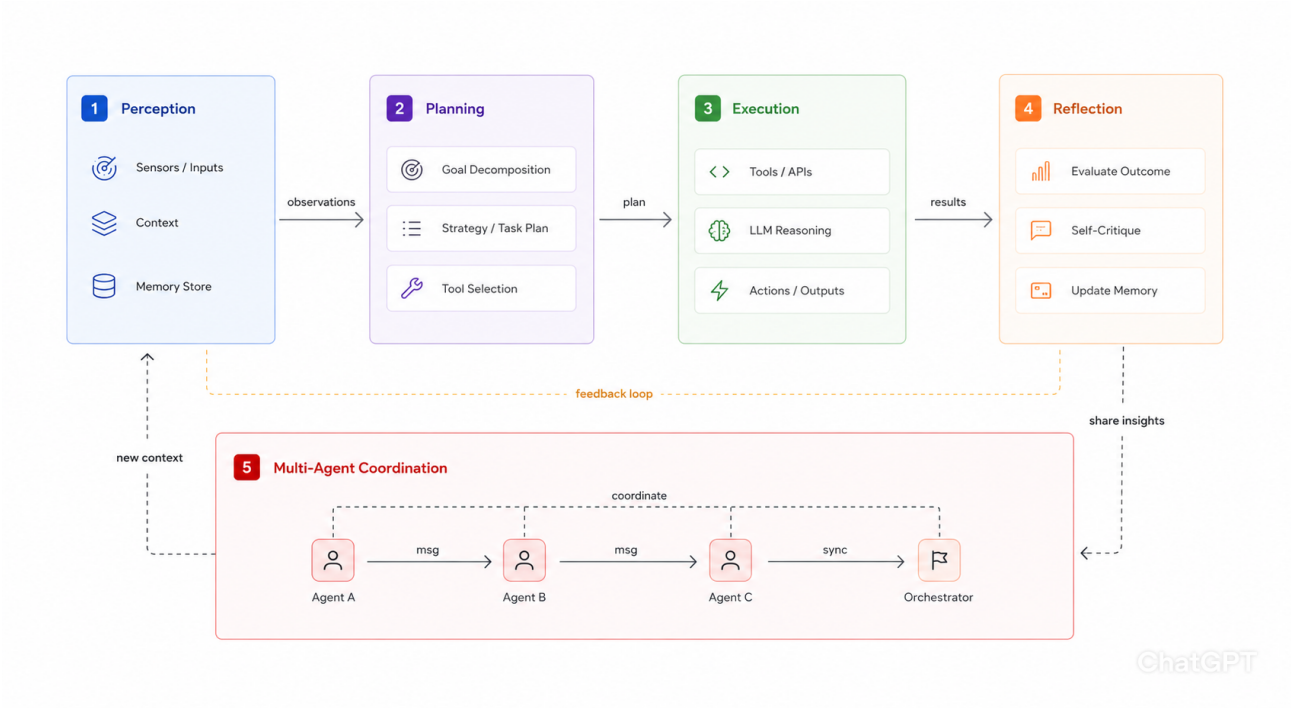


Figure 2.4: Agentic AI workflow and orchestration

2.6. Conclusion

The technical foundations underlying modern AI-assisted systems, including language models, Transformer architectures, and agent-based frameworks, define how such systems represent information, model dependencies, and interact with external tools. Their probabilistic behavior, dependence on context, and reliance on external systems highlight the need for validation and control mechanisms, especially in software engineering where correctness and reliability are critical.

Chapter 3 Code Review and Real-World Applications

3.1. Introduction

Code review plays a central role in modern software development by supporting code quality, collaboration, maintainability, and defect detection throughout the development lifecycle. Recent advances in artificial intelligence have introduced new possibilities for automating and assisting review activities through AI-based review systems integrated into real-world development environments. This chapter examines current code review practices and their limitations, analyzes representative industrial AI review solutions, and presents the problem addressed in this work together with the proposed solution.

3.2. Code Review in Software Development

3.2.1. Role of Code Review

Code review is a core software engineering practice used to evaluate source code before integration into shared repositories. Its objectives include defect detection, enforcement of coding standards, maintainability improvement, and knowledge sharing across team members [40].

Beyond bug detection, review also contributes to architectural consistency, security validation, performance awareness, and long-term project sustainability.

3.2.2. Pull Request-Based Review Workflow

In modern collaborative environments, code review is typically performed through pull request workflows. Developers implement changes in isolated branches, open a pull request, receive feedback, revise the code, and merge after validation.

This process improves traceability and collaborative quality control, but its effectiveness depends on reviewer availability, review depth, and the size of incoming

changes.

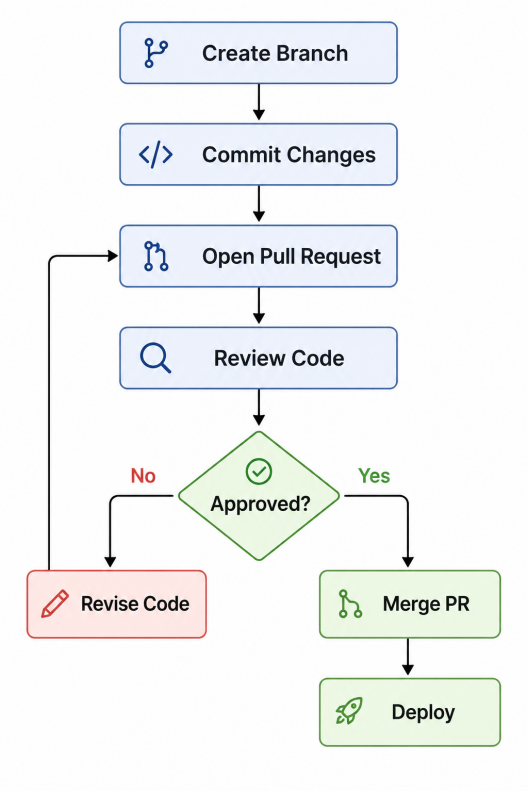


Figure 3.1: Traditional pull request-based review workflow

3.2.3. Tool Support and Operational Limits

Review workflows are supported by linters, static analysis, automated tests, CI/CD pipelines, and security scanners. These tools improve consistency and automate repetitive checks, but they mainly focus on syntactic and rule-based validation.

As AI-assisted coding increases the frequency and size of pull requests, teams face reviewer fatigue, tighter review windows, and higher cognitive load. Under these conditions, traditional review pipelines become harder to scale while preserving consistent quality.

3.2.4. Operational Best Practices

To keep review quality high without creating delivery bottlenecks, common practices include:

- keeping review sessions bounded (for example by time or change size);

- splitting large changes into smaller, incremental pull requests;
- distributing review requests across available domain experts;
- writing constructive comments that include technical rationale.

3.3. Real-World AI Code Review Applications

3.3.1. Greptile

3.3.1.1. Definition

Greptile is an AI code review agent that automatically reviews pull requests with full repository context. Rather than treating changed files as isolated diffs, it builds a graph of the codebase including functions, classes, and dependencies, then evaluates how a change affects the wider system [41].

3.3.1.2. How It Works

Greptile follows a repository-aware review pipeline:

- **Repository connection and indexing:** after integration with GitHub or GitLab, Greptile maps repository structure and dependency relationships.
- **Automatic PR analysis:** for each pull request, it analyzes the change with full repository context and posts review comments with suggested fixes.
- **Agent-assisted remediation:** review comments can be forwarded through *Fix with your Agent* integrations to coding agents for faster correction.
- **Feedback-driven adaptation:** team reactions and replies (e.g., positive or negative feedback) are used to tune future review behavior and reduce irrelevant comments over time.

This allows Greptile to detect cross-file issues such as incomplete updates across dependent modules and interface-usage inconsistencies.

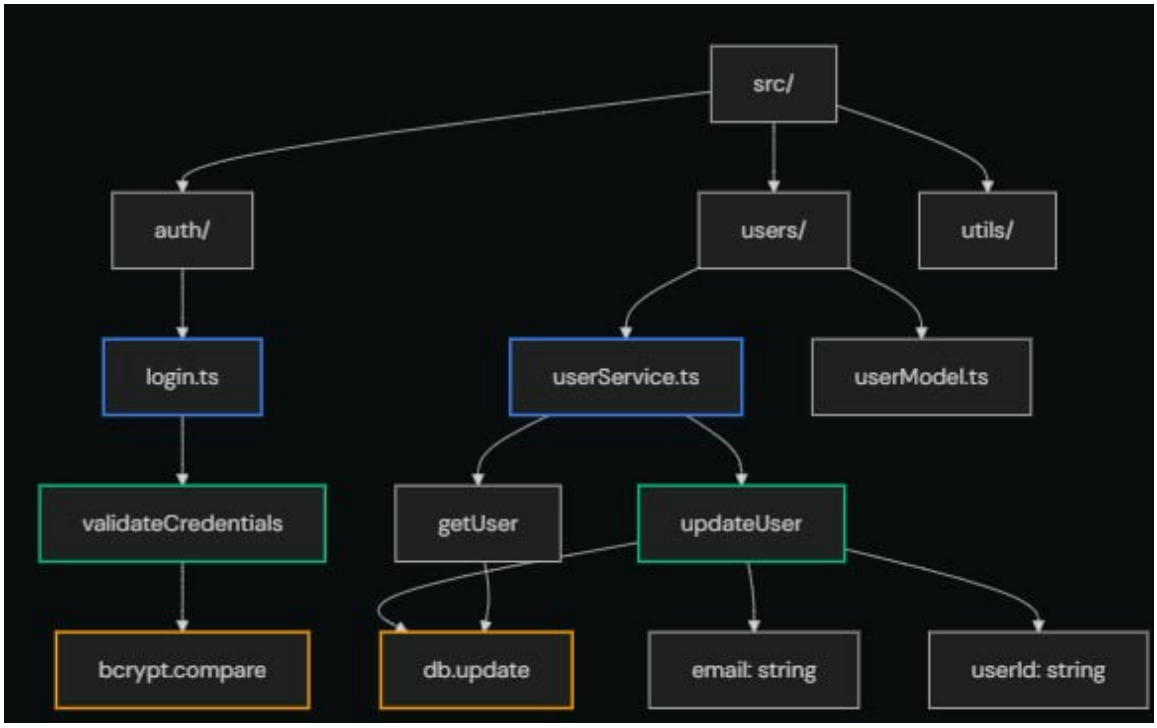


Figure 3.2: Visualized Greptile repository graph construction output [42]

3.3.2. CodeRabbit

3.3.2.1. Definition

CodeRabbit is an AI-powered pull-request review platform that provides automated, context-aware feedback on code quality, security, and maintainability. It combines LLM-based reasoning with static analysis integrations to review proposed changes within repository context [43].

3.3.2.2. How It Works

CodeRabbit operates as an event-driven review pipeline on pull requests [43]:

- **Automatic triggering:** when a pull request is opened, CodeRabbit starts a full analysis of the proposed changes.
- **Multi-layer analysis:** it combines AI reasoning with integrated linters and security scanners to detect potential issues and produce structured feedback.
- **Review publication:** within minutes, it posts PR comments, generated summaries or walkthroughs, severity-labeled findings, and one-click fix suggestions.

- **Incremental re-review:** when new commits are pushed to the same PR, it performs focused incremental reviews on newly introduced changes.
- **Interactive refinement:** developers can interact through PR commands and comments; team feedback is incorporated over time to improve suggestion relevance.

In agent-assisted workflows, this supports an iterative cycle of review, automated correction, and re-review.

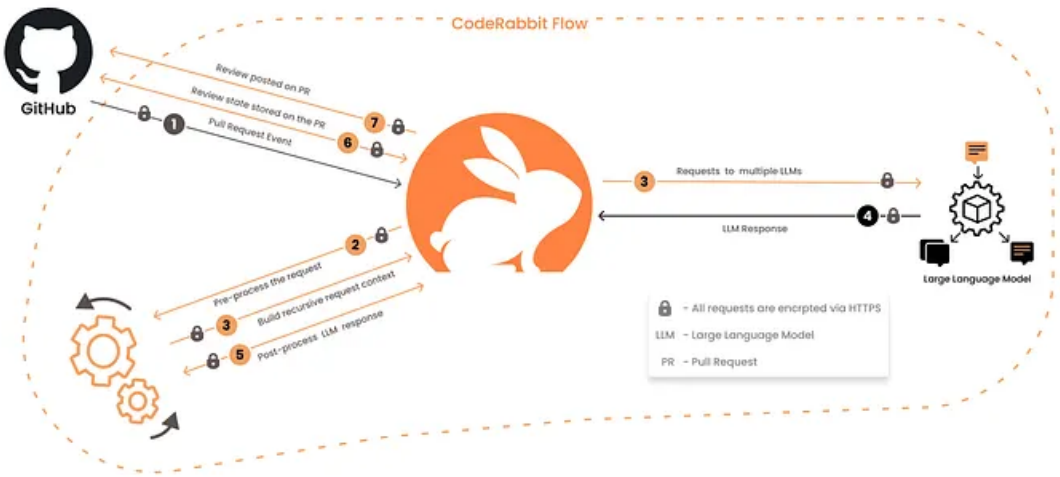


Figure 3.3: CodeRabbit workflow [44]

3.3.3. Vercel AI Agent

3.3.3.1. Definition

Vercel Agent is a suite of AI-powered development tools integrated into the Vercel platform. In the context of code review, it automatically analyzes pull requests and proposes validated fixes, while the broader suite can also support incident investigation using observability signals [45].

3.3.3.2. How It Works

For pull-request review, Vercel Agent follows a validation-first pipeline:

- **Automatic triggering:** reviews start when a pull request is created, when commits are pushed to an open PR, and optionally for draft PRs.
- **Context-aware analysis:** the agent evaluates human-readable files (source code, tests, configuration, and documentation) using full codebase context.
- **Patch generation and sandbox validation:** it generates candidate fixes and runs them in secure sandboxes using the project's real builds, tests, and linters.
- **Validated suggestions only:** only suggestions that pass validation checks are posted in the PR for one-click application.
- **On-demand interaction:** developers can mention `@vercel` in PR comments to trigger review, request fixes, or ask diagnostic questions.

At the suite level, the Investigation capability complements PR review by analyzing logs and metrics around anomaly alerts to support faster root-cause analysis.

3.3.4. Cubic AI

3.3.4.1. Definition

Cubic AI is an AI-powered code reviewer for GitHub pull requests. It is designed to identify bugs and meaningful code improvements in complex repositories while keeping review feedback concise and actionable [46].

3.3.4.2. How It Works

Cubic AI operates through an automated pull-request review and refinement loop:

- **GitHub App integration:** teams install the Cubic GitHub App on selected repositories.
- **Automatic PR review:** when a new pull request is opened, Cubic analyzes the proposed changes and posts inline comments focused on defects and improvements.
- **Configurable standards:** custom agents and team rules can be used to enforce project-specific coding conventions.

- **Feedback adaptation:** the reviewer improves over time by learning from developer feedback on review outcomes.
- **Agent-assisted remediation:** background agents can automatically implement fixes in a sandboxed environment.

For pull requests created before installation, review can be triggered manually through a PR comment command.

3.3.5. RovoDev

3.3.5.1. Definition

RovoDev Code Reviewer is an enterprise-grade LLM-based code review automation system developed and deployed at Atlassian. It is designed as a review-guided, context-aware, and quality-checked assistant that generates pull-request comments in Atlassian Bitbucket while preserving a human-in-the-loop review process [1].

3.3.5.2. How It Works

RovoDev follows a staged generation-and-filtering pipeline for pull requests:

- **Context collection:** when a pull request is created, the system gathers diff data, PR title and description, relevant files, and linked Jira issue information.
- **Zero-shot review generation:** using structured prompts and review guidelines, it generates candidate review comments without fine-tuning on customer code.
- **Factual-correctness check:** an LLM-as-a-judge component filters out comments that are not well grounded in the specific code changes.
- **Actionability check:** a quality gate filters vague or non-actionable comments so that only comments likely to drive code resolution are retained.
- **Comment publishing:** validated comments are posted back to the pull request for developer review and discussion.

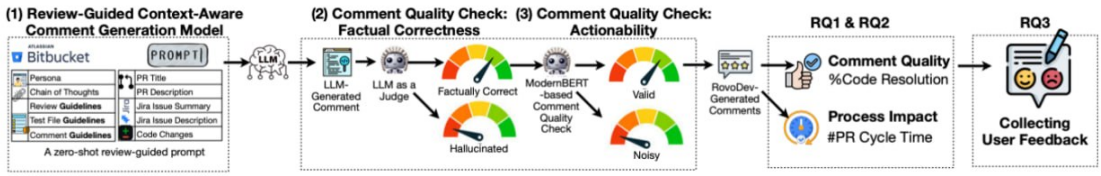


Figure 3.4: RovoDev workflow [1]

3.4. Benefits and Limitations of AI-Assisted Code Review

3.4.1. Benefits

Across the analyzed industrial applications, AI-assisted code review provides several recurring advantages [47], [48]:

- **Faster review and delivery cycles:** automated PR analysis provides early feedback and reduces waiting time before human review.
- **Improved defect and security detection:** AI reviewers can identify bug patterns, regressions, and common vulnerability classes that may be missed under time pressure.
- **Higher consistency of standards:** automated checks apply naming, quality, and policy rules more uniformly across teams and repositories.
- **Better scalability with growing code velocity:** as PR volume increases, automated reviewers absorb repetitive checks and reduce review bottlenecks.
- **Reduced reviewer fatigue:** by handling routine findings, AI tools allow human reviewers to focus on architectural and domain-specific decisions.

3.4.2. Limitations

Despite these benefits, important constraints remain and must be considered in practice [47], [48]:

- **False positives and review noise:** low-priority or incorrect alerts can reduce trust and increase triage effort.

- **Limited business-context understanding:** many tools still struggle with project-specific intent, product constraints, and non-local requirements.
- **Stochastic and uneven output quality:** model-based suggestions can vary across runs and may miss edge cases.
- **Configuration and adoption overhead:** teams need initial setup, rule tuning, and feedback loops to achieve reliable signal quality.
- **No replacement for human accountability:** final decisions on correctness, architecture, and trade-offs still require expert human review.

In practice, the most effective strategy is a hybrid workflow: AI systems provide rapid, scalable first-pass analysis, while human reviewers perform high-level validation and final approval.

3.5. Comparative Analysis

Across these systems, a common architectural pattern emerges: agent-driven iteration over code with increasing levels of contextual awareness. All analyzed tools support PR reasoning to some extent, but they differ in how context and feedback are integrated.

3.6. Conclusion

The examination of code review practices and AI-assisted review systems highlights the growing role of artificial intelligence in supporting software quality, validation, and collaboration within modern development workflows. The analysis of existing industrial solutions showed that current tools remain specialized, improving specific review and validation tasks without fully connecting code changes, runtime behavior, and automated remediation processes. These observations establish the problem addressed in this work and justify the need for a more integrated and intelligent review approach.

Chapter 4 Design, Implementation, and Evaluation

4.1. Introduction

Building a production-grade AI platform requires translating architectural decisions into concrete implementations that integrate across multiple components and workflows. This chapter presents the design, implementation, and evaluation of the proposed system. The chapter covers the architectural design and rationale for each component, the monorepo-based implementation, complete end-to-end workflows, an experimental evaluation, and a discussion of challenges and limitations.

4.2. Problem Statement

The widespread adoption of AI-assisted coding tools has significantly increased the speed and volume of software production. While this improves developer productivity, it also creates new challenges for maintaining software quality, **reliability**, and security.

AI-generated code can still contain logical errors, insecure implementations, architectural inconsistencies, or maintainability issues that require human validation. However, manual review processes are constrained by reviewer availability, limited review time, and increasing cognitive workload.

As pull requests become more frequent and complex, existing review workflows struggle to scale effectively. This creates a need for intelligent systems capable of assisting developers in analyzing code changes, identifying risks, and improving review consistency without replacing human oversight.

4.3. Proposed Solution Overview

To address these limitations, this thesis proposes an AI-powered code review system designed to assist developers during pull request evaluation and post-production

incident analysis.

The proposed system integrates into collaborative development workflows and automatically analyzes submitted code changes to generate structured feedback related to:

- Code quality.
- Potential logical issues.
- Maintainability concerns.
- Security risks.
- Best practice violations.

The system is intended to support reviewers rather than replace them, helping reduce cognitive workload while improving consistency and scalability in review processes. In this context, it introduces **observability** by enabling early detection of potential issues through the analysis of code changes, dependency structures, runtime signals, and external incident data, allowing defects, security risks, and performance regressions to be identified before they reach production.

In addition to pull request analysis, the proposed architecture extends into post-production monitoring workflows. Runtime issues detected through monitoring platforms such as Sentry are forwarded to a specialized Mechanic agent capable of analyzing failures, attempting reproduction, generating corrective modifications, and creating pull requests containing proposed fixes. This enhances **reliability** by accelerating issue resolution through automated analysis, structured feedback, and immediate corrective actions integrated into the development workflow.

By combining AI-assisted analysis, automated monitoring, and human supervision, the proposed solution aims to improve software quality assurance across both development and production environments.

4.4. Design and Implementation

4.4.1. Requirements and Design Goals

4.4.1.1. Design Goals

The system was designed around three primary goals. First, it must detect defects, security risks, and maintainability issues at pull request time using codebase-aware analysis rather than isolated diff inspection. Second, it must trace runtime failures captured by Sentry through the dependency graph, generate corrective patches, and open reviewable pull requests, all without fine-tuning on proprietary data. Third, every code modification must pass through human review, with the review pipeline gating normal changes and the remediation pipeline gating automated fixes through Draft pull requests.

Several non-goals bounded the design scope. The system targets TypeScript monorepos exclusively, given that static dependency analysis depends on the TypeScript compiler API. It does not fine-tune language models on customer code, relying instead on prompt engineering and tool-augmented reasoning. It augments rather than replaces human reviewers, enforcing human validation through pull request workflows.

4.4.1.2. Functional Needs

- **GitHub Connection**: The system must connect to a GitHub repository and listen for pull request events to trigger analysis automatically.
- **Pull Request Data Retrieval**: The system must retrieve pull request diffs, metadata, and changed file lists to provide full review context.
- **Dependency Graph Construction**: The system must build a dependency graph from the source code to enable structural understanding beyond flat diffs.
- **Code Change Analysis**: The system must analyze code changes for defects, security vulnerabilities, style violations, and maintainability issues using multi-agent reasoning.
- **Structured Feedback Generation**: The system must produce structured, actionable feedback with file-level evidence, severity levels, and suggested fixes.

- **Comment Posting**: The system must post inline review comments, summary comments, and check run results back to the pull request.
- **Sentry Integration**: The system must integrate with Sentry to receive runtime incident notifications and associated metadata.
- **Draft Pull Request Creation**: The system must open Draft pull requests containing proposed fixes for human review.
- **Data Persistence**: The system must persist reviews, findings, fix records, and connection metadata in a database for auditability and reuse.
- **Authentication**: The system must provide user authentication mechanisms to ensure secure access to dashboard and API endpoints.

4.4.1.3. Non-functional Needs

- **Security**: All user access (both dashboard users and API consumers) must be safeguarded by authentication (Clerk sessions, GitHub App tokens). Agent code execution must be fully isolated from the host system using sandboxed environments.
- **Performance**: The app must be fast and responsive, providing quick loading times for the web dashboard, timely completion of automated PR reviews, and smooth navigation between monitoring, fixes, and alerts views.
- **Scalability**: The architecture must support the growth of users, repositories, and concurrent workloads without degrading performance.
- **Reliability**: The app must be reliable, minimizing errors and system crashes. Automated agents must handle transient failures (API timeouts, rate limits) with retry logic, and webhook endpoints must recover from failures without missing events or losing data.
- **Maintainability**: The system must be built using modular, loosely-coupled components following a monorepo structure, enabling independent development, testing, and deployment of each package. Adding new AI providers, subagent types, or sandbox backends should require minimal changes to exist- ing code.
- **User-friendliness**: The user interface should prioritize intuitiveness, ease of use, and accessibility for all users. Interface elements should be well-organized

and comprehensible, making it easy to navigate reviews, fixes, monitoring, and settings.

4.4.1.4. Use Cases

This subsection identifies the actors that interact with the system and the key functions they perform, providing a high-level behavioral view of the platform before the architectural and implementation details are presented.

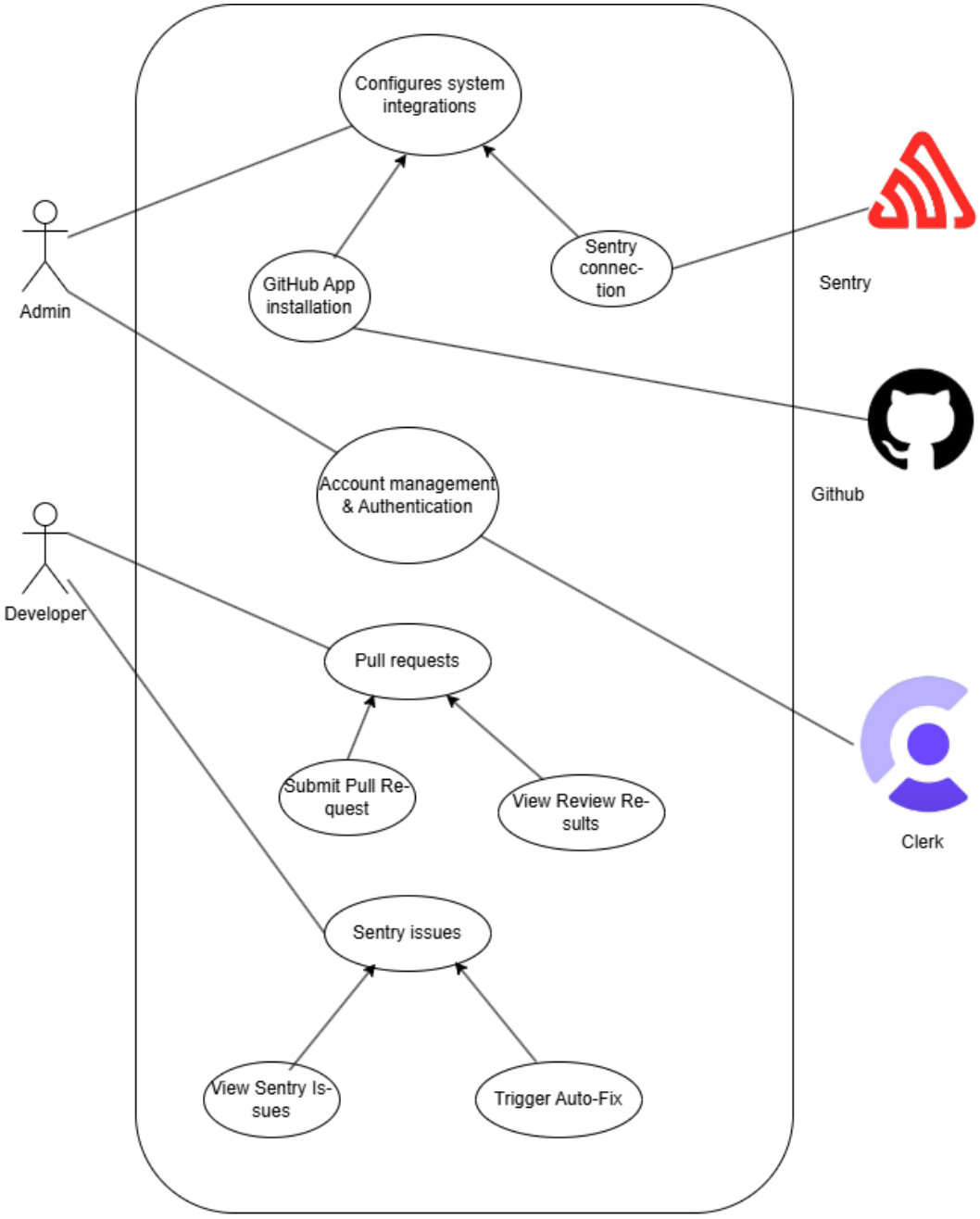


Figure 4.1: Use case diagram

4.4.2. Core Concepts

- **TypeScript**: a statically typed superset of JavaScript that adds optional type annotations and related language features.
- **Webhooks**: event-driven HTTP callbacks used by one system to notify another system when a specific event occurs.
- **GitHub App**: an integration model for connecting external tools to GitHub with scoped permissions and event access.
- **Pull Request**: a proposed set of code changes submitted for review before being merged into a target branch.
- **Dependency Graph**: a graph that represents relationships between elements such as files, modules, or functions.
- **AST**: an abstract syntax tree that models the hierarchical structure of source code.

4.4.3. Technology Stack

The system builds on several key technologies, each chosen for its role in the architecture:

- **Turborepo** and **Bun**: monorepo organization and package management, enabling shared code across agent packages and the web application.
- **Next.js** and **Clerk**: web framework for the dashboard with App Router for API routes, and authentication service for user management and OAuth flows.
- **Prisma** and **PostgreSQL**: database access and persistence for reviews, findings, Sentry connections, and fix runs.
- **ts-morph**: TypeScript compiler API wrapper used by the codebase graph engine for parsing, symbol resolution, and dependency extraction.
- **Octokit**: official GitHub SDK for API interactions including pull request data retrieval, comment posting, and check run management.
- **Zod**: schema validation library that enforces structured output from language models, rejecting malformed findings at runtime.

- **Vercel AI SDK**: framework for building tool-augmented agents, handling model integration, tool execution, and structured output generation.
- **Sentry**: error tracking platform that supplies runtime incident data (stack traces, breadcrumbs, request context) to the mechanic agent.
- **Vercel Sandbox**: ephemeral execution environments where agents operate in isolated clones of the target repository.

4.4.4. Monorepo Package Overview

Falcon is implemented as a TypeScript monorepo using Bun workspaces and Turborepo. The workspace is organized into `apps/*` and `packages/*`, with shared task pipelines for `build`, `dev`, `test`, `lint`.

Package Name	**Type**	**Definition**	**Role**
Web application	Application	Next.js dashboard and API application	Hosts UI, webhook entrypoints, review endpoints, monitoring endpoints, and integration routes.
Codebase graph	Infrastructure	Static TypeScript dependency graph engine	Builds graph snapshots and serves structural queries (callers, impact, dependencies).
Sandbox manager	Infrastructure	Provider-agnostic sandbox lifecycle manager	Provisions isolated execution environments, runs commands, and enforces retry/reconciliation logic.
GitHub API	Infrastructure	Typed GitHub App integration library	Handles pull request retrieval, check runs, comments, and repository operations.
New review agent	Agent	Multi-agent pull-request reviewer	Executes specialist review agents, orchestrates findings, and returns schema-validated results.
Mechanic agent	Agent	Autonomous remediation agent	Investigates Sentry incidents, applies fixes, verifies changes, and opens Draft pull requests.
Review agent (legacy)	Agent (legacy)	Earlier reviewer generation	Retained for compatibility and reference against the newer reviewer pipeline.
OpenCode Go provider	Provider	LLM provider adapter	Supplies model access and fallback routing for agent execution.
Better Copilot provider	Provider	GitHub Copilot-compatible provider adapter	Provides primary model transport for review and summarization paths.
GitHub SDK provider	Provider	SDK integration provider layer	Bridges provider-specific SDK calls used by model backends.
TypeScript config	Shared configuration	Shared TypeScript compiler baseline	Centralizes reusable TypeScript settings to keep package configuration consistent.

Table 4.1: Monorepo package structure and roles

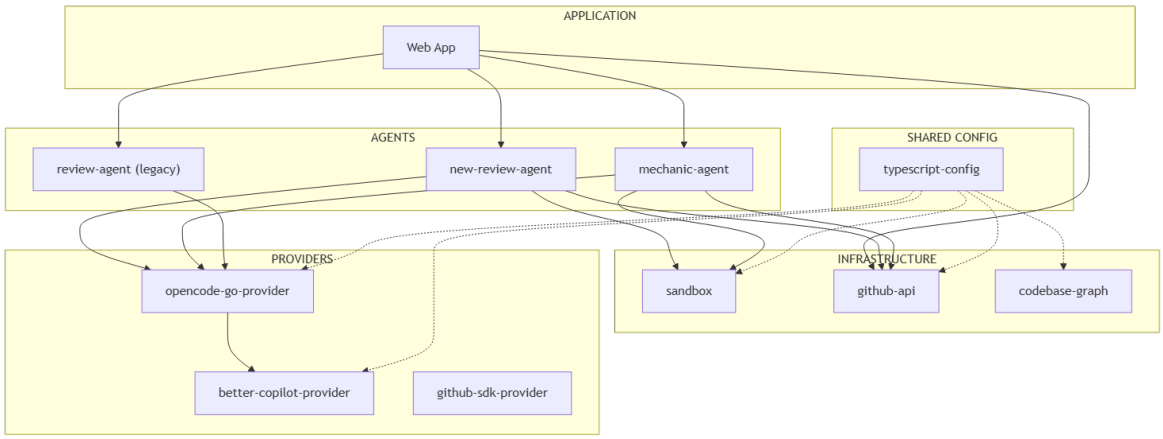


Figure 4.2: Monorepo package dependency graph showing runtime dependencies between workspace packages

4.4.5. System Architecture Overview

Falcon follows a pipeline architecture with two distinct but related flows sharing a common infrastructure layer. The review pipeline processes pull request events from GitHub webhooks, provisions an ephemeral sandbox, loads or builds a codebase dependency graph, summarizes the diff, and routes the context to a multi-agent reviewer that produces structured findings with file-level evidence. The remediation pipeline is initiated by a user action on a synced Sentry issue; it provisions a sandbox, loads the graph, and runs a single autonomous agent that investigates the failure, applies a patch, verifies it through linting and type-checking, and opens a Draft pull request.

Both pipelines depend on three shared subsystems. The codebase graph engine provides structural awareness beyond flat diffs, enabling agents to reason about caller-callee relationships, cross-package dependencies, and impact boundaries. The sandbox layer provides ephemeral, isolated execution environments where agents can read files, run searches, execute commands, and apply changes without affecting the host system. The GitHub integration layer handles webhook reception, authenticated API client management, check run lifecycle, and comment posting. A PostgreSQL database persists repositories, reviews, findings, Sentry connections, and fix run records across both pipelines.

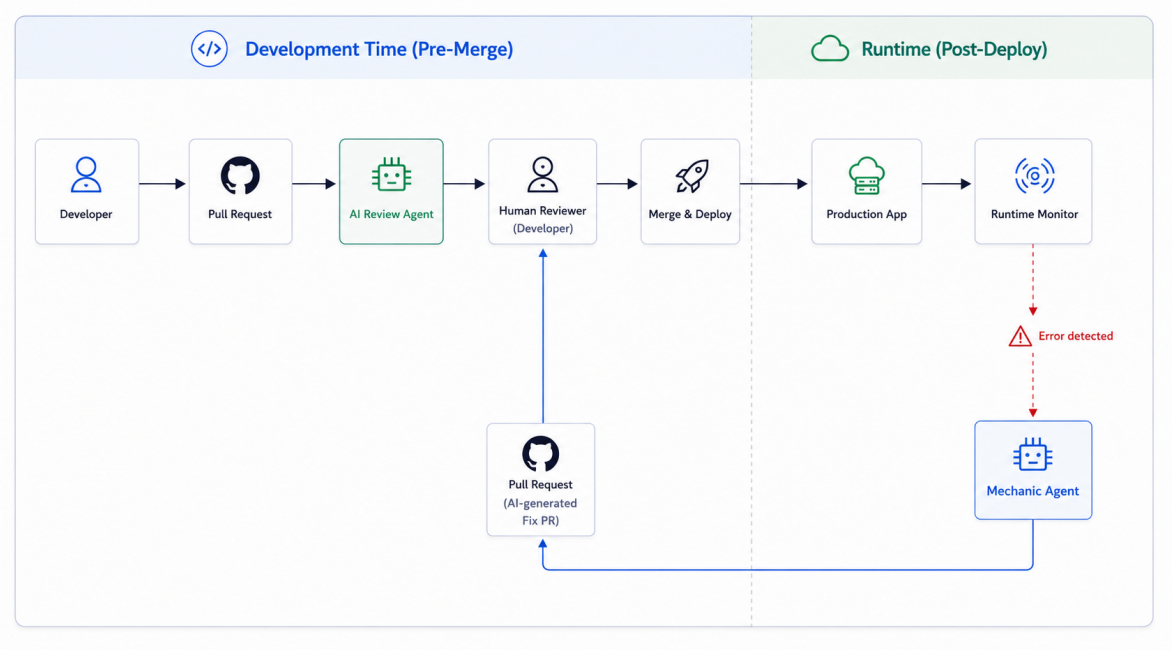


Figure 4.3: Overview of the proposed AI-powered code review system

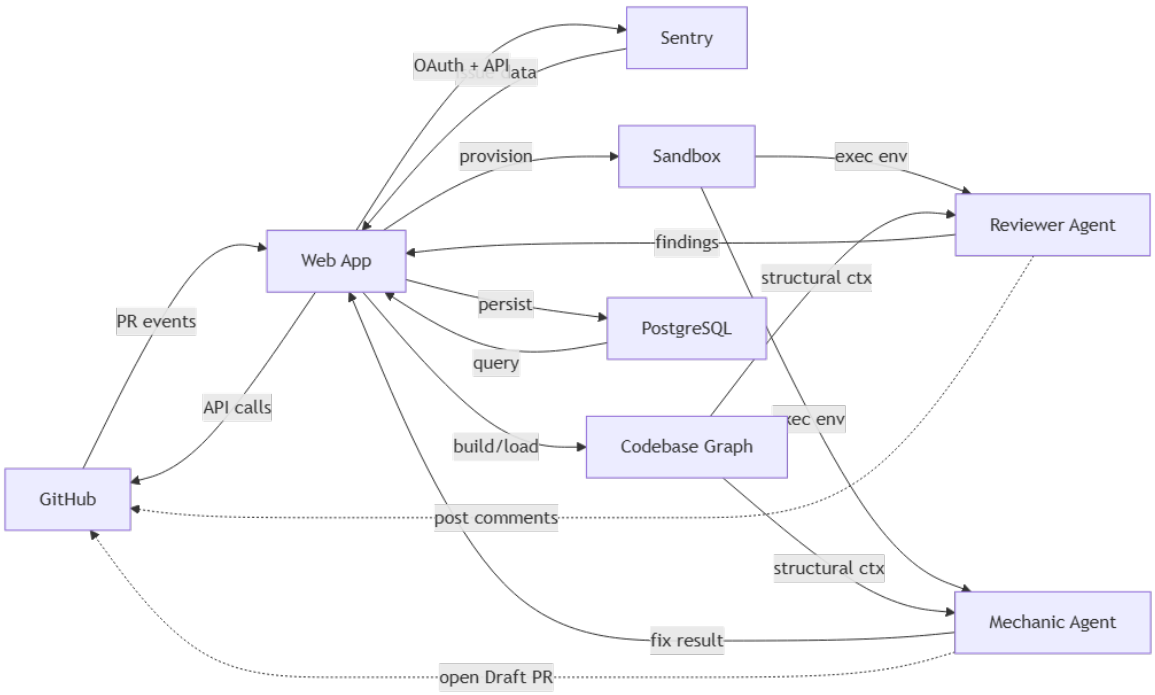


Figure 4.4: System-wide data flow showing interactions between external services, core components, agents, and persistence layer

4.4.6. Codebase Graph Engine

The codebase graph engine constructs a static dependency graph from TypeScript source files before agent execution begins. It represents the codebase as a set of nodes connected by directed edges. Nine node kinds model the structural hierarchy and the declarations within it: the workspace, its packages, individual files, and the declarations they contain (functions, methods, classes, interfaces, type aliases, and variables). Eight edge kinds capture how these nodes relate: containment (a file belongs to a package), imports (one file references another module), function calls, variable usage, class inheritance, interface extension, type references, and cross-package dependencies. In practical terms, this means the graph can answer questions such as "which functions does this file call?" and "which packages depend on this module?" without scanning source text at query time.

Construction follows a two-pass ordering. Pass one registers all source files, extracts declarations (function signatures, class definitions, interface contracts, type aliases, and variable statements), and resolves import edges by following each import statement to its target file or external module. Pass two resolves cross-reference edges. Call edges are identified by locating every `CallExpression` node in the AST and resolving its target through TypeScript's symbol resolution. Variable usage edges link `Identifier` references to their corresponding declarations. Type and inheritance edges capture class extension, interface implementation, interface extension, and type reference relationships. This two-pass ordering is necessary because pass two cannot resolve a call target or a variable reference until pass one has registered the corresponding declaration node in the graph.

Workspace discovery reads the root `package.json`, resolves workspace glob patterns, and creates a package record for each discovered workspace member. File scanning walks each package recursively while excluding build artifacts and metadata directories such as `node_modules`, `.git`, `.turbo`, and `dist`. Path alias resolution parses `tsconfig` `compilerOptions.paths` mappings, which allow developers to define custom import shortcuts; the engine tries relative path resolution first, then alias matching, then probes candidate file extensions before recording any unresolvable import as an external edge rather than failing.

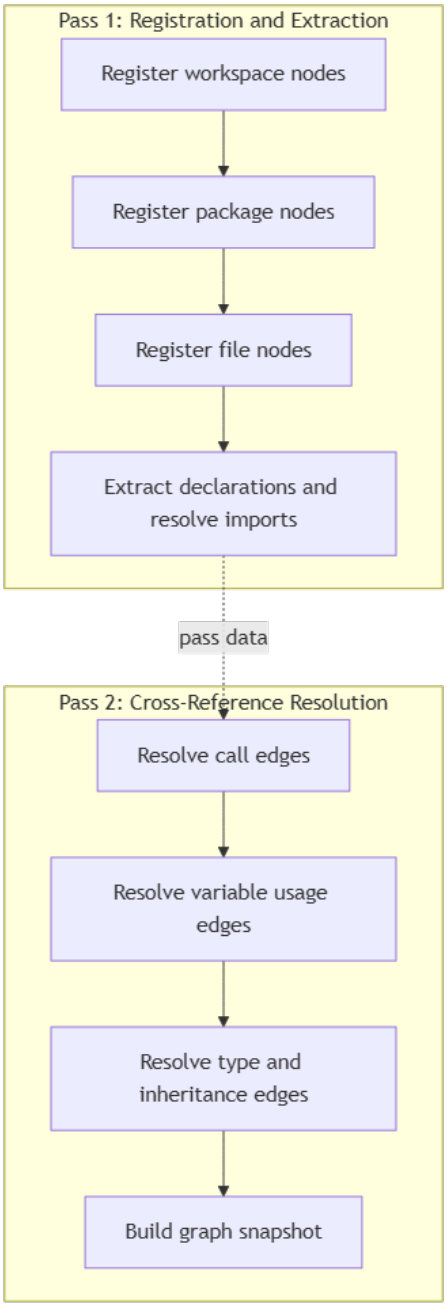


Figure 4.5: Two-pass graph construction: Pass 1 registers nodes and extracts declarations; Pass 2 resolves cross-references

The graph is designed to support structured queries that agents can use during investigation, rather than requiring them to traverse raw source files. Each query addresses a specific class of structural question:

- `findCallersOf(function)` returns all functions that call a given function, enabling an agent to trace impact across call chains rather than guessing which files to inspect.

- `findImpactOf(file)` performs a breadth-first traversal along incoming edges (imports, calls, containment) to compute the blast radius of a change, meaning the set of files that may need modification when a given source file is altered.
- `findUnusedFunctions()` identifies functions that have no incoming call edges and are not exported, flagging candidates for dead code removal without requiring manual grep-based searches.
- `findCrossPackageDependencies()` collects all edges that cross workspace package boundaries, providing agents with visibility into inter-package coupling that is invisible from a single package's source tree alone.

These queries give agents structural context that a flat diff cannot provide.

4.4.6.1. Implementation Realization

```
Algorithm: Build Dependency Graph
Input: workspace root path
Output: graph snapshot

context <- initContext(root)
registerWorkspaceNodes(context)
registerPackageNodes(context)
registerFileNodes(context)

for each file in context.files do
resolveImportEdges(context, file)
extractDeclarations(context, file)
end for

for each file in context.files do
resolveCallEdges(context, file)
resolveVariableUsageEdges(context, file)
resolveTypeAndInheritanceEdges(context, file)
end for

return buildSnapshot(context.graph)
```

4.4.7. Sandbox Layer

The sandbox layer provides ephemeral, isolated execution environments for agent workflows. The architecture follows a provider-agnostic control plane with two abstractions: a provider contract that defines the backend operations (create, get, list, stop, extend timeout, run command) and a manager component that handles lifecycle, state tracking, startup reconciliation, and retry logic. Decoupling the provider contract from the manager allows the system to support different sandbox backends without modifying agent code.

Sandbox lifecycle is modeled with six explicit states:

- `pending`: provisioning in progress, sandbox not yet ready
- `running`: sandbox is active and ready for command execution
- `stopping`: shutdown has been initiated
- `stopped`: sandbox has been terminated
- `failed`: an error occurred during provisioning or execution
- `unknown`: the state cannot be determined from the provider

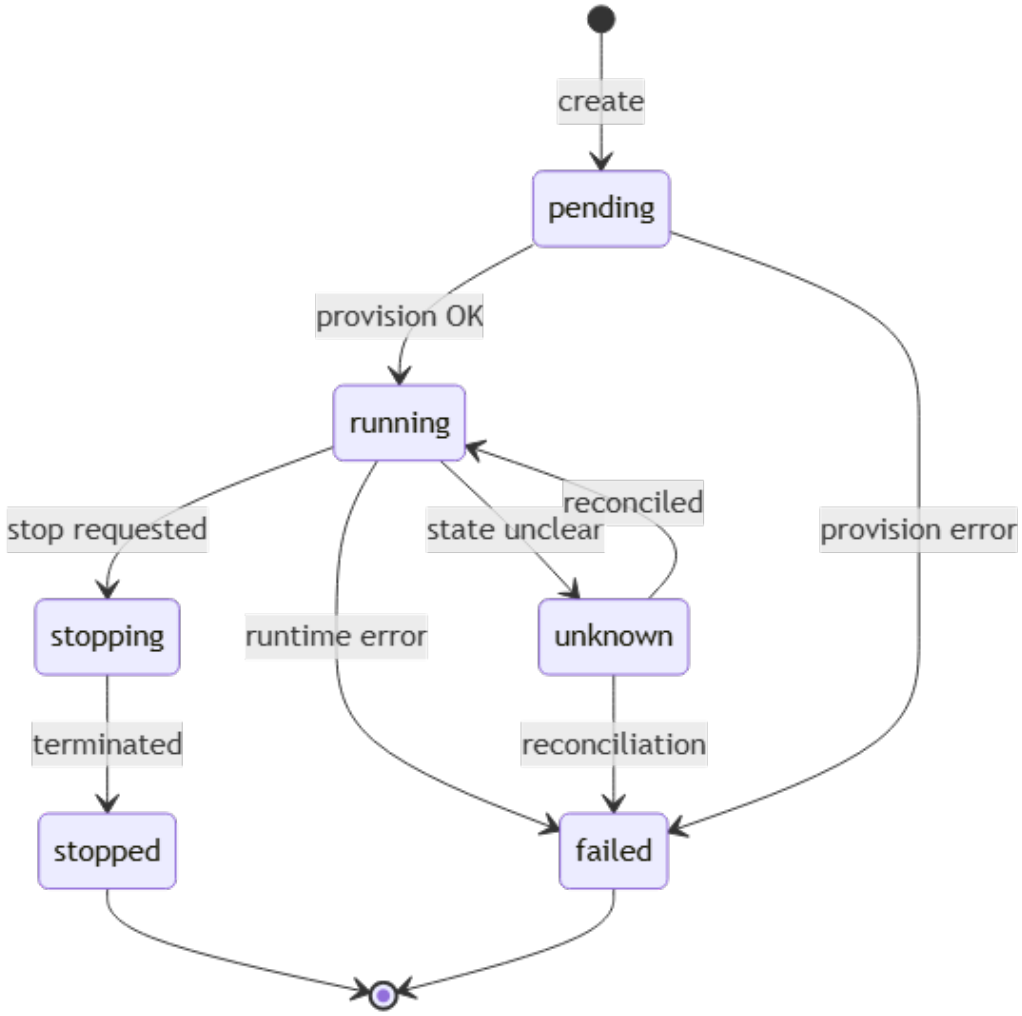


Figure 4.6: Sandbox lifecycle state machine with six states and transition conditions

State is maintained in an in-memory map and, optionally, in a persistent store. The provider is treated as the authoritative source of truth during normal operation, with the store serving as a fallback for crash recovery.

A startup reconciliation mechanism prevents stale state after crashes. On initialization, the manager queries the provider and store for active records: provider-reported sandboxes take precedence, while store-only records missing from the provider are marked as stopped.

Bounded retries with exponential backoff and jitter handle transient failures (rate limits, timeouts, HTTP 5xx); non-retryable errors (authentication, not-found, configuration) are surfaced immediately through typed error classes. Each sandbox is provisioned from git at the target revision, commands execute with working-directory and timeout constraints, and the environment is stopped after completion regardless

of outcome.

4.4.7.1. Implementation Realization

```
Algorithm: Initialize and Reconcile Sandbox State
Input: active sandboxes from provider, active sandboxes from
   store
Output: consistent in-memory and persisted state

providerIndex <- indexById(providerActive)

for each sandbox in providerActive do
persisted <- findById(storeActive, sandbox.id)
merged <- mergeProviderAndStore(sandbox, persisted)
upsertMemoryState(merged)
persistState(merged)
end for

for each sandbox in storeActive do
if not exists(providerIndex, sandbox.id) then
corrected <- markStopped(sandbox, "missing-in-provider")
upsertMemoryState(corrected)
persistState(corrected)
end if
end for
```

```
Algorithm: Execute Operation with Retry
Input: operation, retry policy
Output: operation result or normalized failure

lastError <- null

for attempt <- 1 to maxAttempts do
try
return operation()
catch error
lastError <- error
if (not isRetryable(error)) or (attempt = maxAttempts) then
throw normalizeError(error)
end if
delay <- backoffWithJitter(attempt)
```

```
16  sleep(delay)
17  end try
18  end for
19
20  throw normalizeError(lastError)
```

4.4.8. Reviewer Agent

The reviewer agent is a tool-augmented, schema-constrained, multi-agent LLM system. Its objective is to produce actionable code-review findings with file-level evidence. The agent consumes three artifacts: the raw pull request diff (the primary evidence source), a precomputed codebase graph (for structural context), and an optional diff summary (advisory only, never treated as ground truth).

4.4.8.1. Multi-Agent Architecture

The architecture combines eight parallel specialist agents with a final orchestration stage, improving recall across defect classes compared to a single general-purpose agent.

The review pipeline follows five stages. First, repository context preparation resolves branch state, computes changed files, and prioritizes review targets. Second, shared context assembly builds a common context object for all workers, including changed files, the precomputed diff, branch metadata, and an optional summary, with an explicit contract that the raw diff is authoritative. Third, eight domain-specialized sub-agents run concurrently with bounded tool-use budgets and structured output constraints. Fourth, a dedicated orchestrator agent merges, deduplicates, validates, and reprioritizes findings. Fifth, if parallel execution produces no valid signal (for example, every sub-agent's output fails JSON parsing and the fallback parsing pipeline also fails), a single-agent fallback executes using a dedicated general review prompt (separate from the eight specialist prompts) and a subset of the available tools.

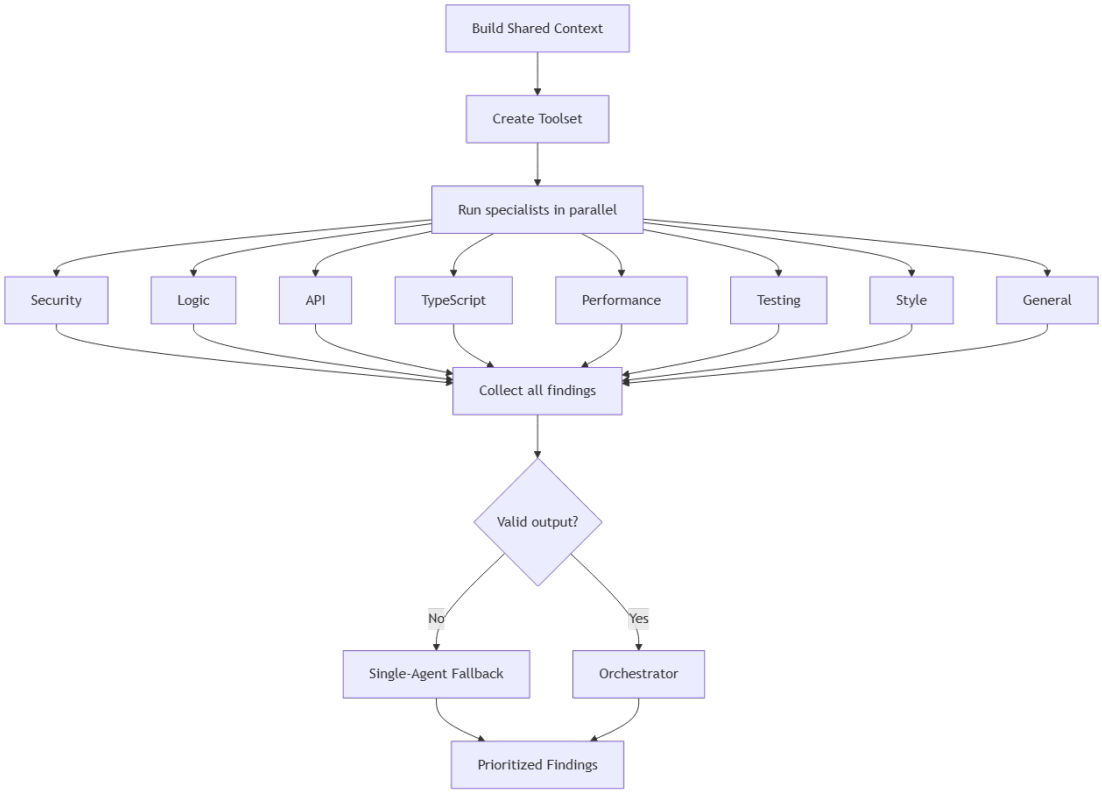


Figure 4.7: Reviewer multi-agent pipeline: context assembly, parallel specialist execution, validation, orchestration, and fallback

4.4.8.2. Two-Phase Prompting Strategy

Each sub-agent follows a two-phase reasoning protocol:

1. **Cold Read**: the agent inspects the full diff without tool access, forming an independent interpretation of intent, behavioral change points, and likely risks.
2. **Explore Mode**: the agent uses tools to validate or reject those hypotheses by reading files, searching symbols, and querying graph relationships.

This separation reduces early anchoring on noisy retrieval and enforces evidence-driven verification before findings are reported.

4.4.8.3. Specialized Sub-Agents and Roles

Eight specialists cover distinct defect classes, each with calibrated exploration constraints. `minToolSteps` defines the minimum number of tool-using reasoning steps

the agent must complete before it may produce its final output, ensuring a baseline level of investigation. `maxToolSteps` sets a hard upper bound that prevents runaway tool usage and controls cost:

- **Security**: exploitable vulnerabilities, authorization scope errors, injection vectors, and secrets exposure, following OWASP classifications.
- **Logic**: runtime correctness faults including null or undefined failures, off-by-one errors, async misuse, race-prone updates, and unreachable branches.
- **API**: interface and contract integrity, including breaking changes, incompatible request or response shapes, and validation gaps.
- **TypeScript**: type-safety risks such as unsafe assertions, unchecked unknown data, widening `any`, weak narrowing, and type-driven masking of runtime failure.
- **Performance**: scalability and efficiency regressions including unbounded queries, missing pagination, N+1 patterns, and redundant work.
- **Testing**: verification adequacy including missing tests for changed behavior, weak edge-case coverage, flaky patterns, and integration gaps after contract changes.
- **Style**: hygiene and maintainability, including dead code, unused paths, excessive complexity, and readability debt, typically flagged as non-blocking.
- **General**: broad cross-domain review for issues not captured cleanly by specialist prompts, serving as a final sanity-check layer.

Each sub-agent executes as one independent LLM call with tool access, structured as follows:

1. **System prompt**: domain prompt (security, performance, logic, etc.), shared context (precomputed diff truncated at 15 000 characters, changed files list, diff summary, branch info, graph context).
2. **User prompt**: You are the X reviewer. Analyze the code diff first, then inspect at least five files using the available tools. The agent must complete 5–8 tool-assisted analysis steps before finalizing the review. Each finding must include the file, line number, and an exact source code quote, while recommendations are only provided when their correctness can be verified with certainty.

3. **Execution**: The system operates through an LLM-driven loop augmented with analysis tools such as ls, glob, readFile, grep, git, and codebaseGraph. Execution is constrained by a maximum number of tool interactions (maxToolSteps), after which the agent must return its final output in raw JSON format. Throughout the process, tool usage and generated output are logged for monitoring and evaluation purposes.

4. **Parsing**: The output text is processed through a multi-stage parsing pipeline: an initial direct JSON parsing attempt, followed by fallback extraction using regular expressions. Common formatting issues (e.g., trailing commas, single quotes, unquoted keys, and line comments) are corrected during normalization. The resulting structure is then validated against a Zod schema, and any nonconforming entries are discarded. The system also logs the number of dropped items for traceability.

5. **Fallback**: If initial parsing fails, the system triggers a fallback mechanism that re-invokes the LLM without tool access, enforcing a strict "return JSON only" constraint for up to 8 steps. Should this secondary attempt also fail, the system returns an empty set of findings as a safe default.

6. **Output shape**: `{ agentId, findings: [{severity: P0--P4, file, line, quote, title, message, suggestion?}], summary }`.

Domain prompts are adapted according to the defect class. The security-oriented prompt integrates OWASP threat tiers, requires a minimum confidence threshold of 7/10, and applies a phased data-flow tracing approach. The performance-focused prompt uses a structured checklist covering issues such as N+1 queries, missing pagination, bundle bloat, and redundant recomputation. Each domain defines its own severity rules.

4.4.8.4. Tooling Layer

The reviewer operates through six constrained tools executed inside the sandbox, each returning structured success-failure results so the agent can distinguish an empty response from a true error:

- `ls` and `glob`: repository navigation with result limits and depth constraints.
- `readFile`: targeted file or range inspection, using `cat` for files under 120 lines and `sed` line-range extraction for larger files, with token and size safeguards.

- `grep`: code and symbol search using ripgrep with a fallback to `grep -R -n`, with deduplication detection to prevent repeated identical searches.
- `git`: limited to `diff` and `blame` operations; branch manipulation is explicitly disabled.
- `codebaseGraph`: structural queries over the precomputed graph, supporting callers, impact analysis, cross-package dependencies, unused-function detection, and node-level relation details.

4.4.8.5. Diff Summarizer

A compact diff summarizer runs in multiple modes depending on diff size and token budget. For diffs under approximately 50,000 tokens, a single LLM call produces the summary. For larger diffs, chunked mode splits the diff by file blocks then hunks, packs chunks within the token budget, summarizes each independently, then merges the chunk summaries through a reduction step. For very large diffs, a sub-agent orchestrator mode groups files into batches, summarizes per batch concurrently, then performs an LLM-based merge. The summarizer is always advisory; all findings must still be justified by raw diff and inspected code evidence.

4.4.8.6. Output Schema and Severity Model

Reviewer outputs are strictly schema-validated using Zod. Each finding includes severity, file path, line number, an optional verbatim quote, a title, a descriptive message, and an optional suggestion. Severity levels are defined as follows:

- **P0**: production-breaking or security-critical impact (data loss, authorization bypass, exploitable flaw)
- **P1**: behavioral regression or contract violation
- **P2**: latent correctness risk (race-prone or brittle logic)
- **P3**: operational, performance, or observability issue
- **P4**: hygiene or maintainability concern

Findings are ranked by severity during orchestration.

4.4.8.7. Orchestrator

The orchestrator is a separate merge agent that receives all sub-agent outputs and performs six operations: it refines wording and actionability, removes confirmed false positives conservatively, merges duplicates across agents (using a deterministic key of file path, line number, and quote text), recalibrates severities when needed, adds cross-cutting findings revealed only by combining agent outputs, and returns per-agent coverage summaries. If orchestrator output fails schema validation, the system retries up to three times with progressively stricter prompts, then falls back to a deterministic merge that concatenates all findings sorted by severity. Additionally, deduplication is applied deterministically before the orchestrator pass.

4.4.8.8. Reliability and Guardrails

Several mechanisms improve reproducibility and keep outputs machine-consumable under imperfect model behavior. Tool-step budgets bound exploration cost per agent. Duplicate-call suppression in tool executors prevents repeated identical operations. Forced finalization behavior near budget end ensures the agent produces a result even when tool steps are exhausted. Strict JSON contract enforcement, parse validation with retries, and deterministic fallback strategies ensure the pipeline produces structured output regardless of model output quality.

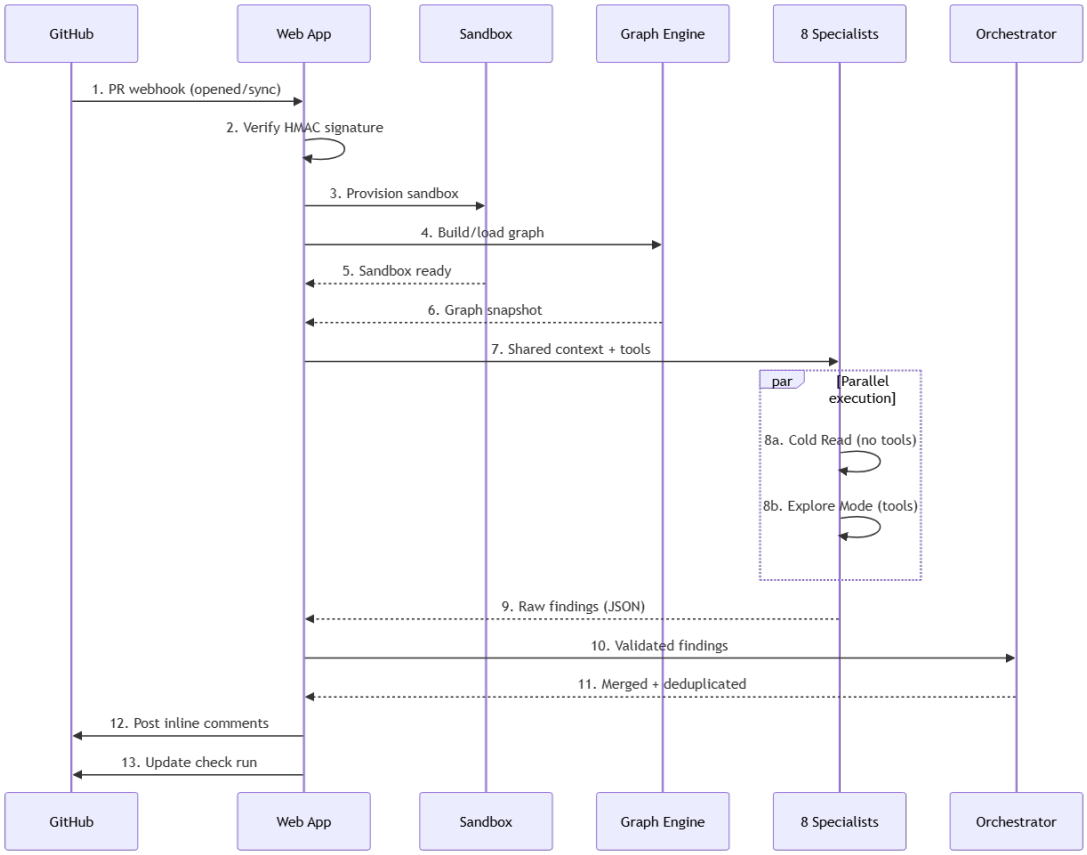


Figure 4.8: End-to-end PR review sequence: from GitHub webhook through sandbox provisioning, parallel specialist execution, orchestration, and result publication

4.4.8.9. Implementation Realization

```
Algorithm: Run Pull-Request Review
Input: branch context, changed files, diff artifacts
Output: prioritized finding set

sharedContext <- buildSharedContext(branchContext, changedFiles
    , diffArtifacts)
tools <- createToolset(sharedContext)

results <- runSpecialistsInParallel(sharedContext, tools,
    boundedToolSteps)
validated <- validateAgainstSchema(results)

if isEmpty(validated) then
return runSingleAgentFallback(sharedContext, tools)
end if
```

```
14
15  merged <- orchestrateAndDeduplicate(validated)
16  finalFindings <- applyFindingCap(merged.findings)
17  return { findings: finalFindings, summaries: merged.
        agentSummaries }
```

4.4.9. Mechanic Agent

The mechanic agent is a tool-augmented language-model system for post-deployment remediation. Unlike the reviewer pipeline, the current mechanic implementation uses a single autonomous agent rather than parallel specialists, because coherent root-cause analysis and patch generation require maintaining a unified understanding of the codebase state across all investigation steps.

The agent receives structured incident metadata from Sentry: issue title, severity, occurrence count, affected users, culprit, permalink, and timestamps, plus optional latest-event payload data including exception type and message, stack frames with code context, breadcrumbs, and request metadata. This gives the agent a failure-focused starting point that grounds its investigation in operational evidence.

The agent follows a five-stage reasoning protocol enforced through its system prompt:

1. **Understand**: reads the Sentry context and explores relevant code areas using read and search tools.
2. **Find Root Cause**: traces callers through the codebase graph, examines edge cases, and evaluates null-safety and boundary conditions to form a hypothesis.
3. **Write the Fix**: applies a minimally scoped patch using edit or write tools.
4. **Verify**: runs lint and typecheck commands inside the sandbox, iterating up to three attempts if verification fails.
5. **Report**: returns a schema-validated fix result containing a summary, root-cause explanation, verification status, verification notes, a list of changed files with descriptions, and a confidence flag.

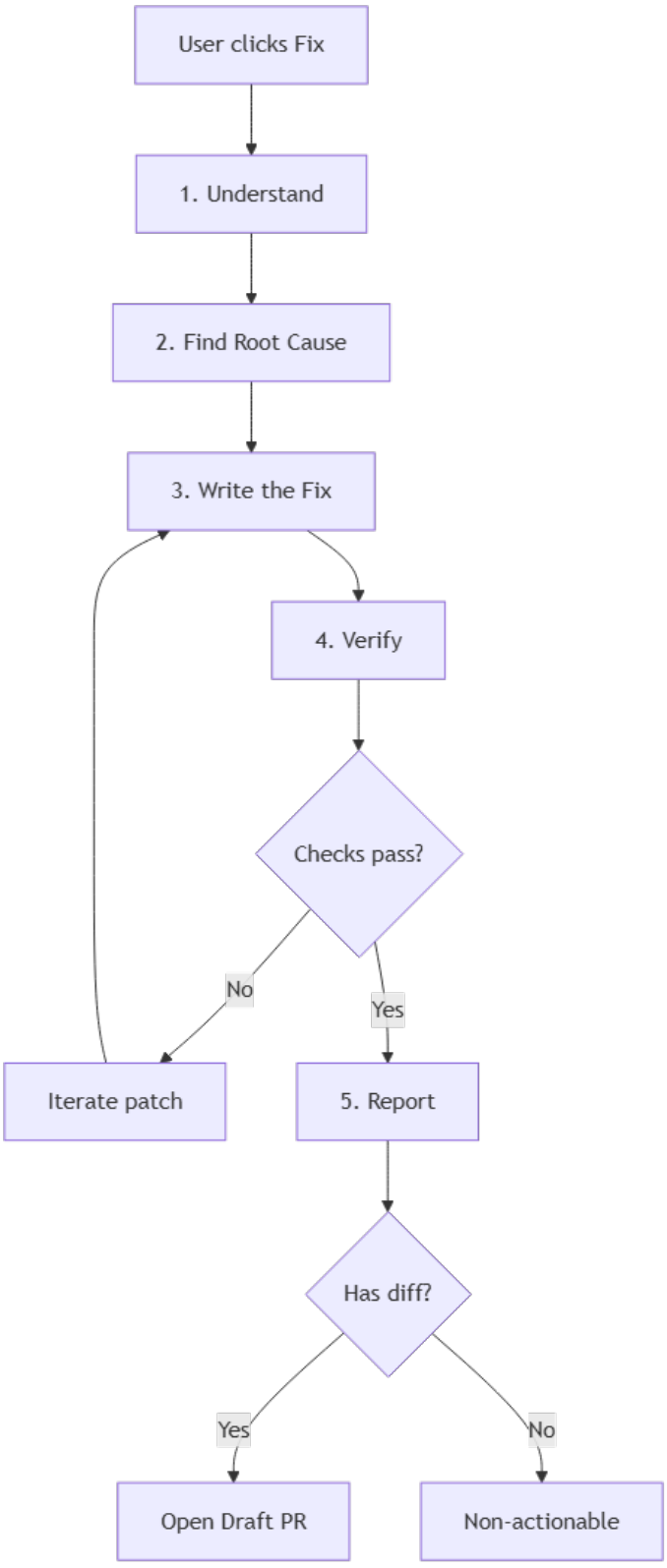


Figure 4.9: Mechanic agent five-stage protocol with verification retry loop and Draft PR creation

If no diff is produced, the run is non-actionable and no PR is opened. Otherwise, the

system creates a branch `fix/sentry-<shortId>`, commits, pushes, and opens a Draft PR containing the Sentry reference, root-cause explanation, fix summary, changed files, and verification outcome.

4.4.9.1. Implementation Realization

```
Algorithm: Run Sentry-Guided Remediation
Input: Sentry issue context, repository context
Output: fix result and optional Draft pull request

incidentPrompt <- buildIncidentPrompt(issueContext,
    eventMetadata)
fixResult <- runAutonomousFixer(incidentPrompt,
    repositoryContext)

if isEmpty(gitDiff()) then
return nonActionable(fixResult)
end if

branch <- createFixBranch("fix/sentry-<shortId>")
commitPatch(branch, fixResult)
pushBranch(branch)
draftPr <- openDraftPullRequest(branch, fixResult)
return actionable(fixResult, draftPr)
```

4.4.10. GitHub Integration

An authenticated event-driven layer bridges repository events to review execution and result publication. Webhook requests are verified by signature before dispatch. Pull-request events trigger review execution, comment publication, check-run updates, and artifact persistence.

4.4.10.1. Implementation Realization

```
Algorithm: Process GitHub Webhook Request
Input: HTTP request with GitHub event payload
Output: accepted event response or validation error

```

```
secret <- loadWebhookSecret()
headers <- readRequiredHeaders(request)

if isMissing(secret) or isMissing(headers) then
return validationErrorResponse()
end if

rawPayload <- readRawBody(request)
if not verifyHmacSignature(rawPayload, headers.signature,
   secret) then
return unauthorizedResponse()
end if

event <- parseEventPayload(rawPayload, headers)
handlerResponse <- dispatchEvent(event)
return handlerResponse or successAck()
```

```
Algorithm: Handle Pull-Request Event
Input: validated pull_request event
Output: review publication and persisted review record

prData <- fetchPullRequestData(event)
diff <- buildUnifiedDiff(prData.changedFiles)

reviewResult <- runReviewPipeline(prData, diff)
placement <- mapFindingsToDiffHunks(reviewResult.findings, diff
   )

publishInlineComments(placement.inline)
publishSummaryComment(placement.unplaced)
updateCheckRun(reviewResult.status)
persistReviewAndFindings(prData, reviewResult)

return reviewResult
```

4.4.11. Data Model Design

The data model is organized into three groups: shared account scaffolding, review pipeline state, and remediation pipeline state, all persisted in PostgreSQL via Prisma.

Shared scaffolding connects users to repositories. A `User` links to `GithubInstallation` records (representing GitHub App installations), each associated with multiple `Repository` records.

Review pipeline produces `Review` and `Finding` records. A `Review` stores the full PR analysis result with a unique constraint on repository + PR number for idempotent re-runs. Each `Finding` includes severity, file location, content, and a `postedToGitHub` flag to distinguish published from database-only findings.

Remediation pipeline uses three entities. `SentryConnection` stores encrypted OAuth tokens for Sentry access. `RepositorySentryProject` links a repository to a Sentry project for scoped issue display. `FixRun` tracks each remediation attempt with its outcome, providing an audit trail.

Key design principles include encrypting third-party credentials at rest, using unique constraints to enforce idempotent pipeline execution, cascading deletions to ensure clean removal when a user disconnects a GitHub installation, and separating concerns across model groups so that the review and remediation pipelines can evolve independently without schema conflicts.

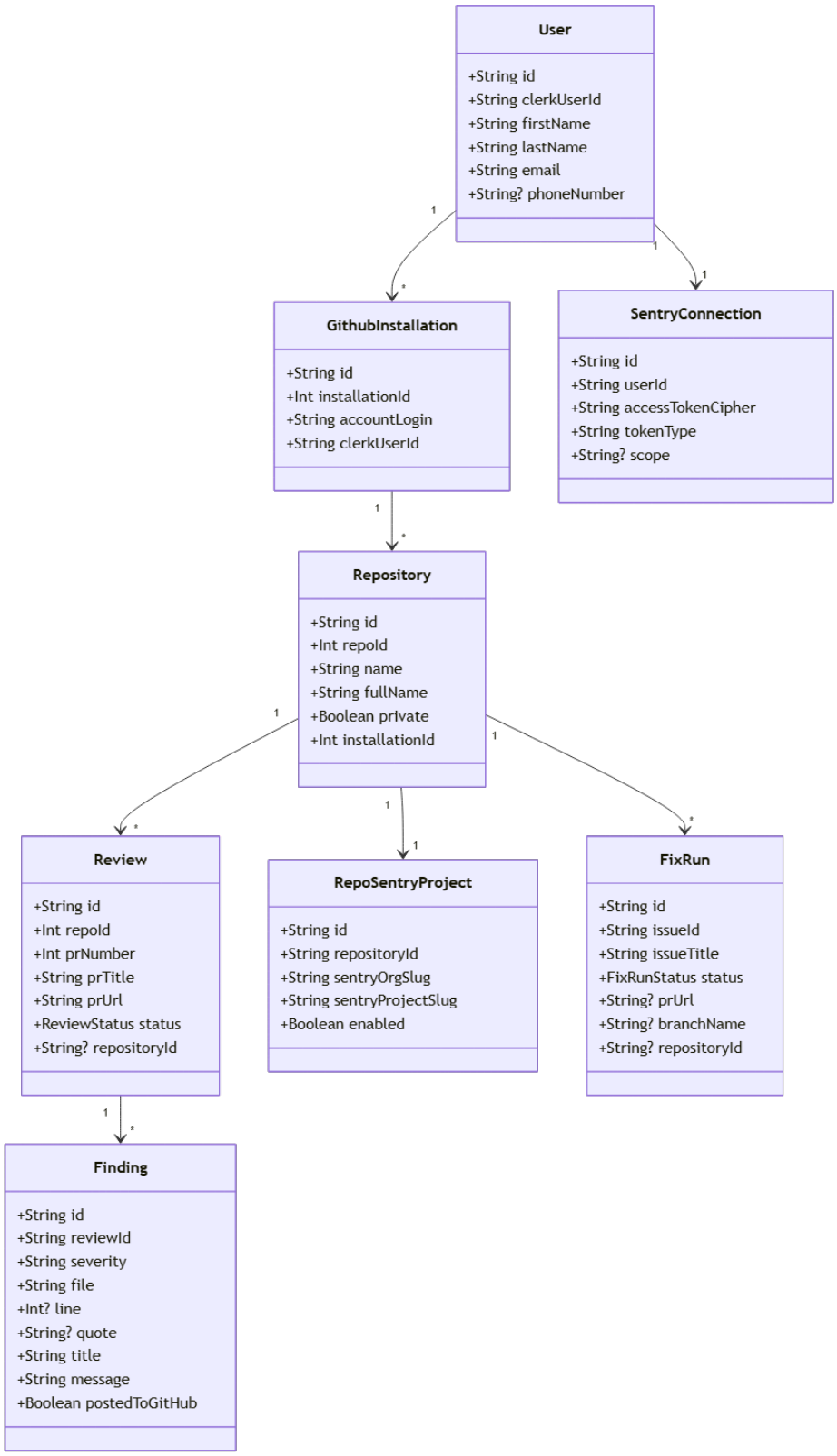


Figure 4.10: Entity-relationship diagram showing the data model organized into shared scaffolding, review pipeline, and remediation pipeline

4.4.12. Web Application Design

The web application provides a dashboard for configuring repositories, viewing review results, monitoring Sentry issues, and triggering remediation actions. Authentication uses Clerk, providing OAuth flows for GitHub and Sentry. The Next.js App Router serves API routes as backend for dashboard data and webhook entry points.

Data-fetching uses SWR for automatic revalidation and Zustand for client-side UI state. API routes are organized by domain: repository endpoints aggregate review counts; review endpoints serve paginated findings with severity breakdowns; monitoring endpoints handle Sentry issue listing and fix triggers; notification endpoints aggregate alerts.

4.5. Application Workflow

4.5.1. GitHub App Installation and Repository Connection

Establishes trusted repository access before any PR analysis or remediation action can start.

1. The user opens the settings page and clicks `Install GitHub App` to start installation.

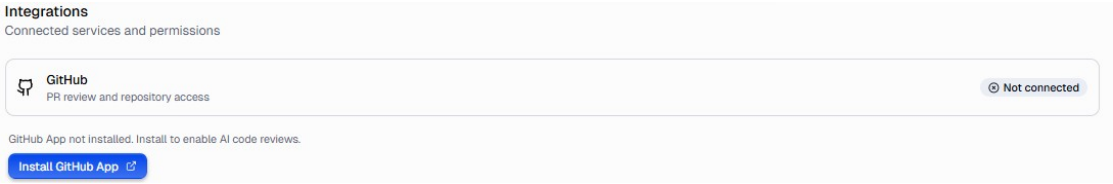


Figure 4.11: Initiate GitHub App installation from settings page

2. The platform redirects the user to the GitHub App installation screen. So the user selects the target personal account or organization where the App will be installed

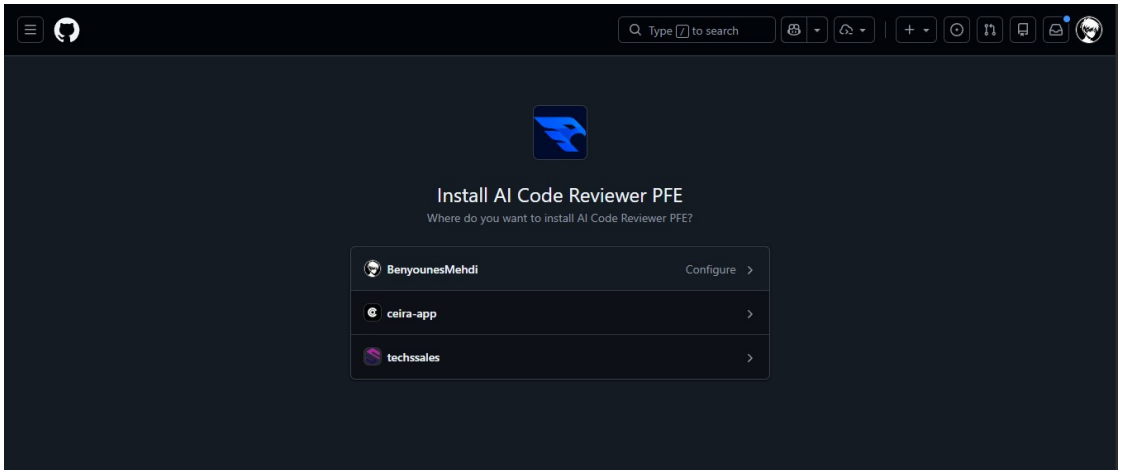


Figure 4.12: Redirect to GitHub App installation

3. GitHub shows requested permissions.

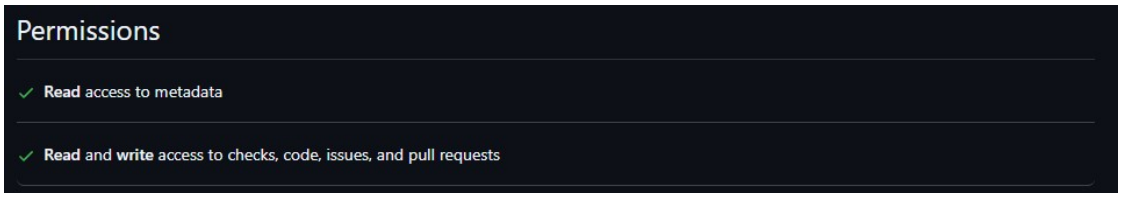


Figure 4.13: GitHub App permissions

4. The user chooses repository scope (all repositories or only selected repositories).

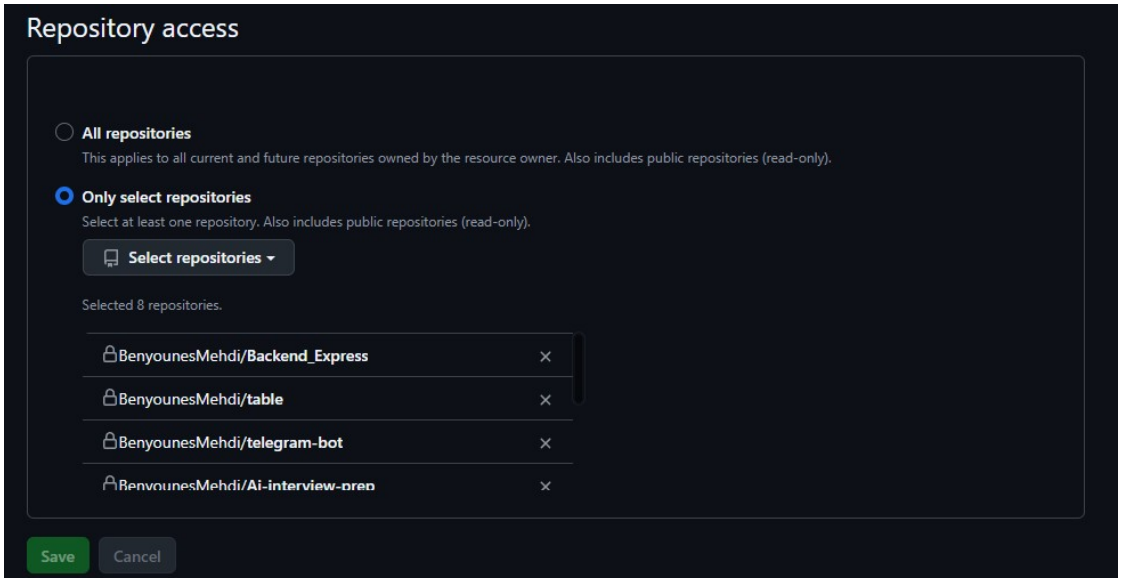


Figure 4.14: Repository access scope selection

5. After confirmation, GitHub redirects back to the platform callback endpoint with installation context.

6. The application validates the installation, stores installation metadata securely, and links the account.

7. Connected repositories are listed in the dashboard and become eligible for automated PR review.



Figure 4.15: Connected repositories visible in dashboard

4.5.2. Code Review Workflow

Once repositories are connected, the review pipeline processes pull request events and generates structured findings with file-level evidence.

1. A pull request event (`opened`, `synchronize`) is received from GitHub.

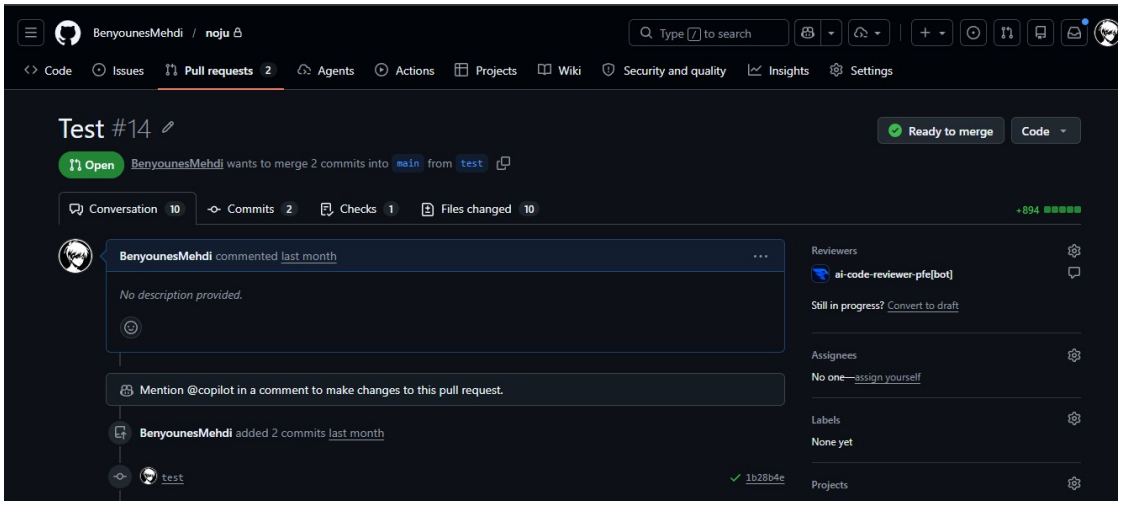


Figure 4.16: Pull request event reception

2. The event signature is verified to ensure the payload came from GitHub.
3. The application extracts pull request metadata, changed files, and the raw unified diff.
4. Parallel jobs start: sandbox provisioning with graph load/build, diff summarization, and check-run initialization.
5. The reviewer agent performs a Cold Read of the entire diff before tool-assisted investigation.
6. The agent enters Explore Mode and queries files, symbols, and graph dependencies to confirm risks.
7. Findings are generated with severity levels and file-line evidence.
8. Findings are posted as annotations and review comments.

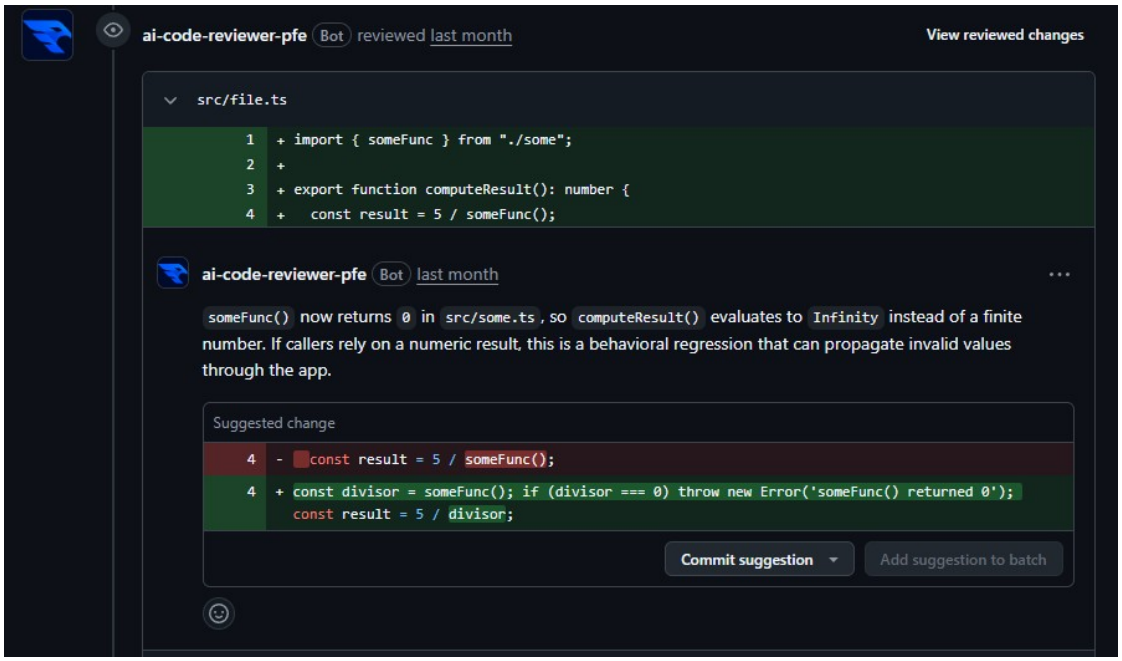


Figure 4.17: Streaming findings as GitHub annotations and comments

4.5.3. Sentry Integration and Issue Sync Workflow

1. The user opens the settings page and selects `Connect Sentry`.

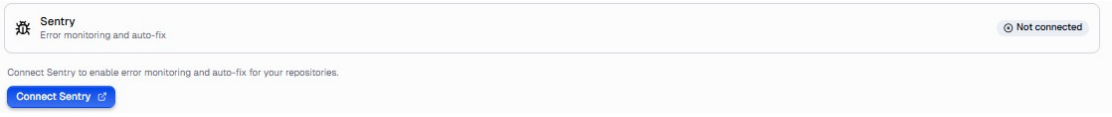


Figure 4.18: Connect Sentry from settings page

2. The user authorizes Sentry workspace and project access for the application.

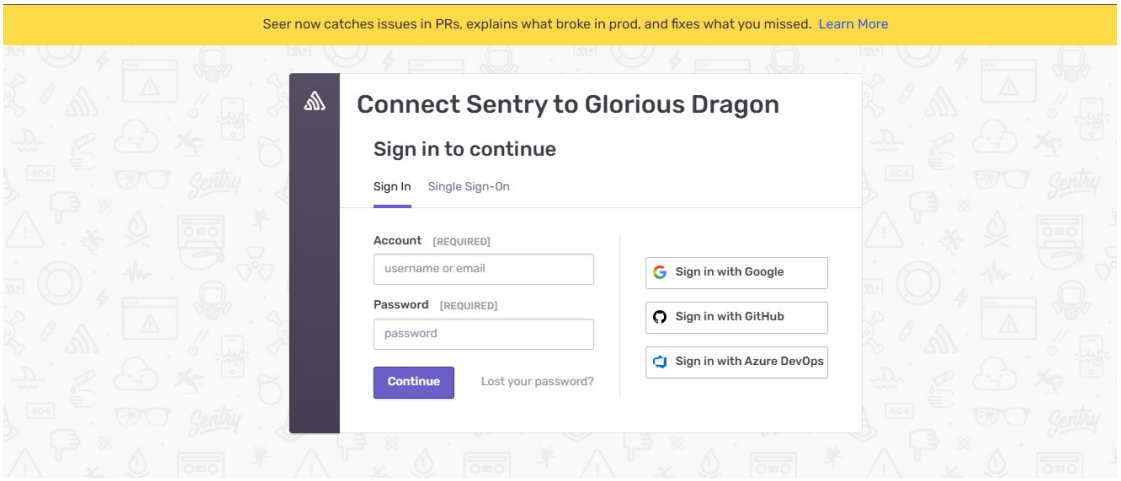


Figure 4.19: Sentry authorization and connection

3. The application stores encrypted Sentry credentials and integration metadata.

4. The application fetches issues from Sentry.

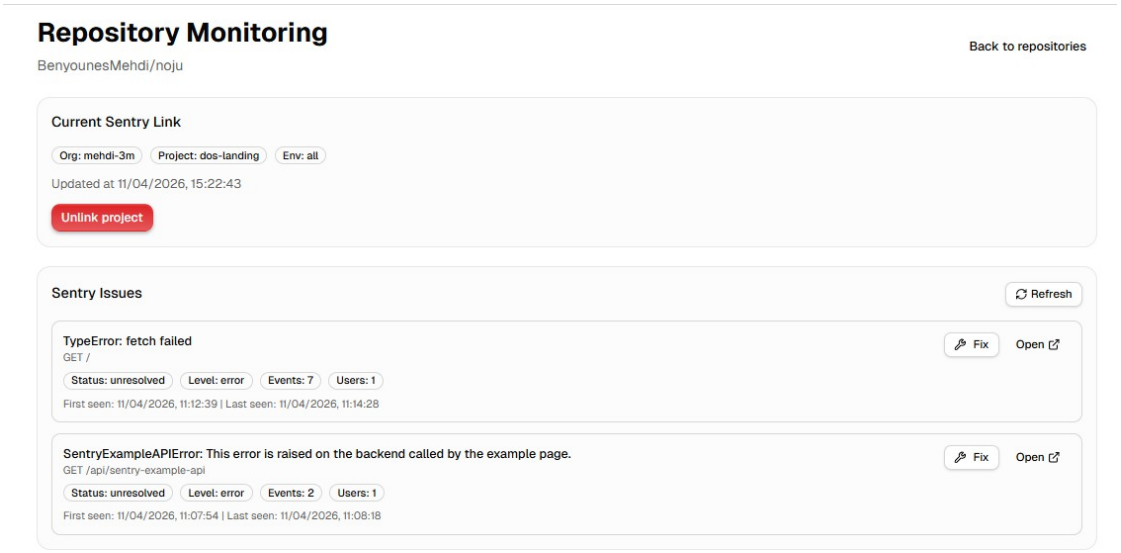


Figure 4.20: Sentry issues fetched and listed in platform

4.5.4. Mechanic Agent Workflow

The mechanic flow is user-triggered from the application. It starts when the user explicitly requests a fix for a synced Sentry issue.

1. From the issues list, the user clicks the `Fix` button on a selected incident.

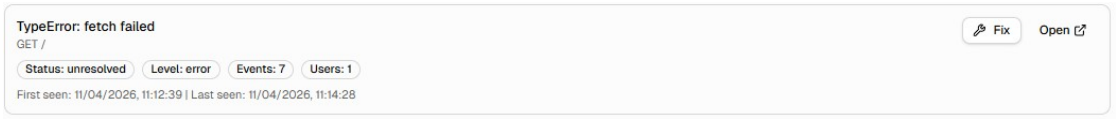


Figure 4.21: User-triggered Fix action

2. The application creates a remediation job and starts the mechanic agent execution session.
3. The agent loads issue context, stack frames, and repository state for the target revision.
4. The investigation phase narrows the probable root cause through stack-trace and graph-guided analysis.
5. The agent generates a patch and applies it in a newly created branch.
6. Validation commands execute in sandbox (tests, type checking, linting) to verify the fix.
7. If validation fails, the agent iterates on the patch until checks pass or the retry budget is exhausted.
8. After success, the application pushes the new branch to the remote repository.

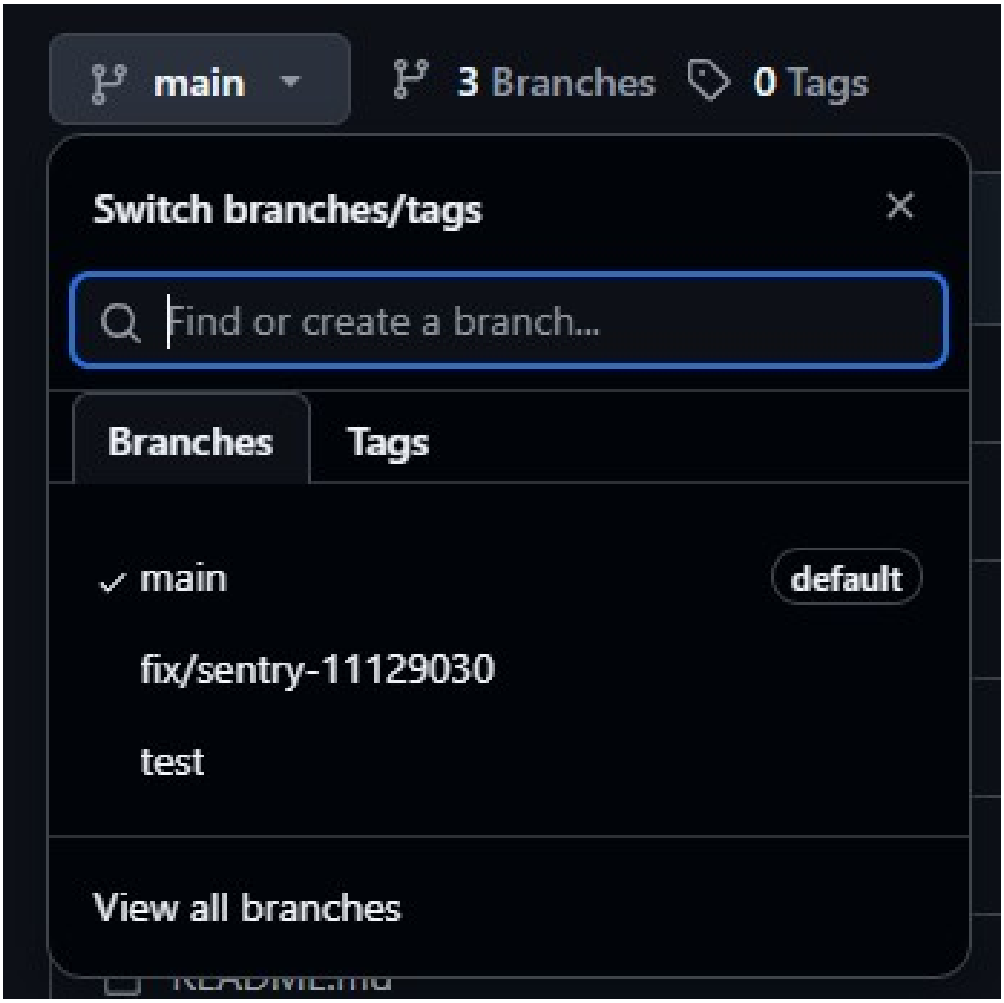


Figure 4.22: Branch creation and publication to remote repository

9. The application opens a new pull request from the mechanic branch to the default base branch to be reviewed by other developers.

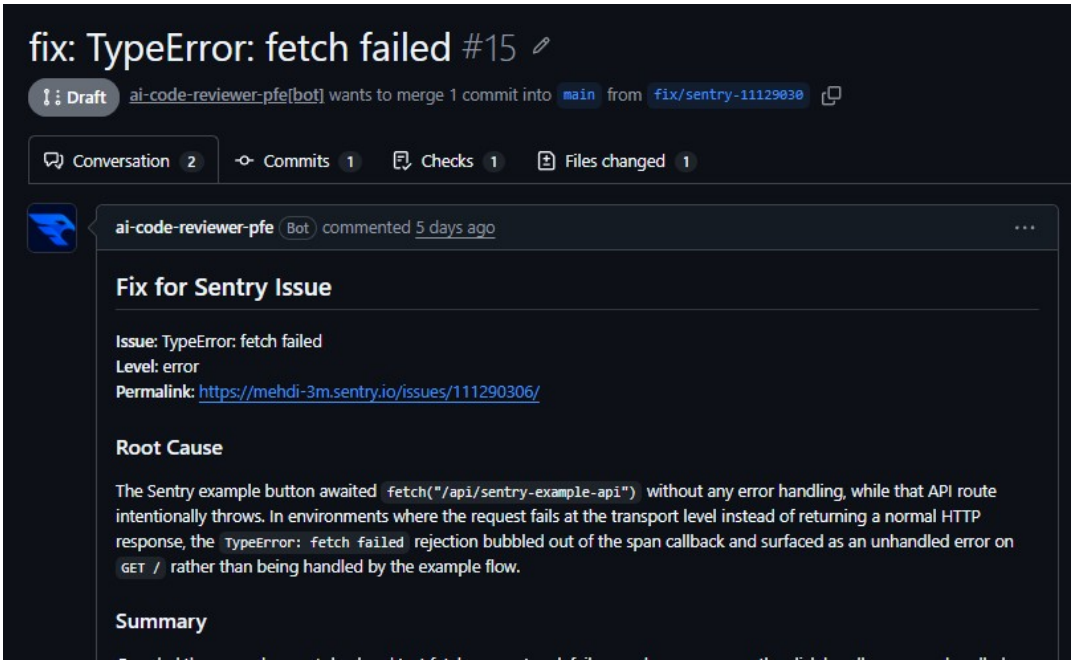


Figure 4.23: Created pull request from mechanic agent

4.6. Evaluation

4.6.1. Evaluation Paradigms for AI Code Review

Several evaluation paradigms have been proposed for assessing automated code review tools. The main options are summarized below.

- **LLM-as-a-Judge:** A language model rates review quality against human references (Qodo Benchmark, CR-Bench). Captures richer context than lexical metrics but is sensitive to prompt wording, model choice, position bias, and stochastic variation.
- **Precision/Recall on labeled datasets:** Review as defect-detection against a ground-truth set. Methodologically attractive but requires complete labels that real workflows do not provide.
- **Human expert panels:** Multiple reviewers score findings for correctness and usefulness. The strongest qualitative approach but costly, time-consuming, and requires inter-rater calibration.
- **Synthetic bug-seeding:** Known defects injected into code, detection rate measured. Controlled but artificial; injected defects may not match real PR

issue distributions.

- **Production metrics:** Acceptance rate, dismissal rate, user retention from deployed systems. Reflects practical utility but requires production deployment and telemetry.
- **Test-based evaluation (c-CRAB):** Human reviews converted to executable tests. A coding agent uses the tool-generated review to revise the patch; passing the test indicates useful feedback. Measures behavioral outcomes but is Python-focused, Docker-dependent, and penalizes tools finding valid issues different from the annotator.
- **Comparative case studies:** Tools compared on the same PR under controlled conditions. Lacks statistical generalizability but enables transparent head-to-head comparison.

4.6.2. Selected Approach and Rationale

For this study, we adopt a comparative case study methodology. The decision is motivated by the constraints of the project and the nature of the system under evaluation.

The test-based paradigm (c-CRAB) represents the current state of the art, but it was not feasible here. Our system targets TypeScript monorepos, while c-CRAB is built on Python repositories. Replicating its curation pipeline (Docker environment construction, LLM-based test synthesis, coding-agent validation) for a TypeScript codebase would require infrastructure and annotation effort beyond the scope of this thesis.

LLM-as-a-Judge was rejected due to reproducibility concerns. Small changes in prompt phrasing can flip evaluation outcomes, and the stochastic nature of LLM outputs makes it difficult to distinguish genuine differences in review quality from random variation.

Precision/Recall requires a ground-truth defect set, which does not exist for the selected pull-request case. Human review comments and post-merge fixes provide only partial labels. Constructing a complete ground truth would require expert annotation and inter-rater agreement procedures that were not available.

Human expert panels and production metrics were logistically infeasible: the former requires multiple paid reviewers with calibrated rubrics, and the latter requires a

production deployment with user telemetry.

Synthetic bug-seeding was considered but deprioritized because injected defects may not represent the distribution of issues found in real PR evolution.

Given these constraints, the comparative case study offers the best trade-off between rigor and feasibility. By evaluating both the proposed agent and Cubic on the same pull request with identical inputs, we control for task difficulty and isolate differences in output quality and execution behavior. The evaluation is transparent (all artifacts, prompts, and raw outputs are logged) and we supplement the comparison with severity and category analysis to move beyond raw findings count.

4.6.3. Evaluation Objectives

Three research questions guide the study:

- **RQ1 (Review Output):** How many actionable findings are produced by each agent, and how are these findings distributed by severity and category?
- **RQ2 (Efficiency):** What execution time is required by each agent?
- **RQ3 (Operational Cost):** For the proposed agent, how many tokens are consumed and what is the estimated monetary cost?

The comparison target is therefore explicit: *Proposed Agent vs. Cubic Review Agent* on the same pull request, using a controlled execution protocol.

4.6.4. Evaluation Setup

4.6.4.1. Pull Request Case Selection

The evaluation is conducted as a focused case study on one real pull request from the `speyar/pfe-monorepo` repository.

PR #99 (`https://github.com/speyar/pfe-monorepo/pull/99`) is included as a representative high-complexity case because it combines schema evolution, API changes, UI modifications, and agent-pipeline updates.

4.6.4.2. Execution Environment and Agent Configuration

All runs are executed in the same sandbox class to avoid environment-induced variance. The proposed agent uses the same architecture described earlier in this chapter (multi-agent reviewer with orchestration and schema validation). The Cubic review agent is executed through its default review flow under equivalent repository and pull-request inputs.

4.6.4.3. Prompting Protocol and Fairness Controls

To preserve fairness, both systems receive identical pull-request artifacts (title, description, changed files, and unified diff) and are evaluated independently on this pull-request. No post-hoc filtering is applied to one system only. Measurement collection for comparative metrics (time and findings extraction) follows a single logging procedure.

For reproducibility, each run stores: pull request identifier, start and end timestamps, raw output payload, parsed findings, severity labels, category labels, and (for the proposed agent) token and cost metadata.

4.6.5. Metrics

The evaluation reports five primary metrics:

- **Findings Count:** total number of extracted findings for each agent on the evaluated pull request.
- **Review Time:** wall-clock execution duration from review start to final output for each agent.
- **Severity Distribution:** proportion of findings in each severity level (P0--P4).
- **Category Breakdown:** distribution of findings by defect category (security, performance, logic, API, testing, style, and other).
- **Token Consumption and Cost (Proposed Agent Only):** input tokens + output tokens and monetary estimate computed from provider pricing.

The cost computation formulas are:

```
tokens_total = input_tokens + output_tokens
cost_estimate = (input_tokens * input_price_per_token)
+ (output_tokens * output_price_per_token)
```

Because the study is a single-PR case study, no dataset-level averaging is reported.

4.6.6. Results

This section reports comparative results between the proposed agent and Cubic on PR #99 for findings count, severity profile, category profile, and execution time. Token and cost results are reported only for the proposed agent because Cubic token and budget data were not accessible.

Metric	**Proposed Agent**	**Cubic Agent**
Findings count (PR #99)	26	9
Review time (seconds)	185	467

Table 4.2: Comparative metrics on PR #99: proposed agent vs. Cubic

Metric	**Proposed Agent**
Input tokens (PR #99)	405,173
Output tokens (PR #99)	21,721
Total tokens (PR #99)	426,894
Estimated cost (PR #99)	$0.0421

Table 4.3: Operational cost metrics reported for the proposed agent

Token values are measured from the proposed agent execution telemetry on PR #99. Monetary cost is computed from these measured token counts using DeepSeek `deepseek-v4-flash` pricing (cache miss input: $0.14/M, cache hit input: $0.0028/M, output: $0.28/M). For this run, cost is calculated from 254,389 non-cached input tokens, 150,784 cached input tokens, and 21,721 output tokens.

Severity and category analyses are reported as normalized percentages to enable direct qualitative comparison between the two agents.

4.6.7. Comparative Evaluation of Automated Code Review Systems

An LLM-as-a-judge evaluation (kimi-k2.6) was used to compare the two models on Pull Request #99 within the `speyar/pfe-monorepo` repository, with the objective of evaluating the efficacy of each system in identifying code defects, architectural concerns, and quality assurance issues.

The first reviewer, denoted as Cubic, exhibited characteristics consistent with a high-level summarization mechanism. Its contributions were predominantly macroscopic, focusing on contextualizing the Pull Request within the broader codebase evolution. The review provided a comprehensive narrative of the introduced features, migration requirements, and dependency modifications. However, the analysis revealed a marked limitation in microscopic code inspection, with minimal engagement at the line-by-line implementation level.

Conversely, the second reviewer, denoted as Falcon, demonstrated a substantially more granular and systematic evaluation methodology. The system generated approximately twenty discrete, file-specific observations that penetrated multiple architectural layers. Notable characteristics included consistent identification of authorization boundary violations and potential security vulnerabilities, detection of API contract drift and error-handling inconsistencies, recognition of deficient test coverage across newly introduced endpoints, and identification of deduplication logic flaws within the orchestration subsystem. Each observation was accompanied by contextual line references and, in several instances, prescriptive remediation strategies.

Based on the comparative evaluation of review depth, defect detection granularity, and actionable feedback generation, Falcon demonstrates superior performance as an automated code review system for this particular Pull Request.

4.6.8. Discussion

The comparative interpretation focuses on the trade-off between review depth and execution efficiency. A system that reports more findings is not necessarily better unless those findings are balanced across meaningful severities and useful categories. Similarly, lower latency is beneficial only if review coverage remains acceptable.

In practical deployment terms, the preferred system depends on organizational priorities: teams prioritizing risk reduction may prefer broader detection coverage in se-

curity and logic categories, while teams prioritizing response speed may favor shorter review duration. Token and cost interpretations are limited to the proposed agent because equivalent Cubic budget telemetry is unavailable.

4.6.9. Limitations

The evaluation has several limitations that affect the generalizability of the results. The comparative review evaluation is based on a single pull-request case study from one repository, which does not establish statistical significance. Finding quality was validated by a single developer rather than through inter-rater reliability assessment. Token and budget telemetry from Cubic was not accessible, so operational cost comparison is incomplete. The mechanic agent was tested on a single synthetic scenario, which does not represent the diversity of real production failures. No ablation studies were conducted to isolate the contribution of individual design decisions (for example, the impact of the codebase graph on finding quality, or the difference between multi-agent and single-agent review). LLM output is inherently stochastic, and no repeated trials were performed to measure output variance. Without precision, recall, or F1 metrics against a labeled dataset, the finding quality assessment remains qualitative.

These limitations should be weighed against the practical constraints that motivated the chosen methodology, as discussed in the Selected Approach and Rationale subsection. Where more rigorous paradigms (test-based evaluation, precision/recall, human panels) were infeasible, we opted for a transparent comparative case study that enables direct head-to-head comparison under controlled conditions.

4.7. Challenges and Limitations

4.7.1. Project Scope and Limitations

Several capabilities were deliberately excluded from the system's scope. The codebase graph engine supports only TypeScript source files, limiting applicability to repositories that use JavaScript or other languages. The system does not fine-tune language models on proprietary code, relying instead on prompt engineering and tool-augmented reasoning, which constrains the ceiling of domain-specific insight it can achieve. The system is designed to augment rather than replace human reviewers; every code modification passes through a human review step, which means the

system cannot operate in a fully autonomous mode. There is no formal security verification of the review or remediation pipelines themselves, and the system has not been deployed in a production monitoring capacity beyond the evaluation repository.

4.7.2. Architectural Limitations

The TypeScript-only graph engine is the most significant architectural constraint. Projects that mix languages, use JavaScript without type annotations, or rely on dynamically resolved imports will produce incomplete or inaccurate graphs. The mechanic agent's single-agent design, while appropriate for maintaining coherent reasoning across investigation and patch application, limits its ability to explore multiple hypotheses in parallel. The graph builder has no incremental update capability: even when only a few files change, the entire graph must be regenerated from scratch. Sandbox cold starts introduce latency on first use, particularly when the graph CLI bundle must be downloaded and executed. Prompts and tool descriptions are authored in English, which may affect performance on codebases with non-English identifier names or documentation.

4.7.3. Technical Challenges Encountered

Several technical challenges arose during development. Calibrating tool budgets for each sub-agent required iterative experimentation: budgets that were too low prevented adequate exploration, while budgets that were too high consumed excessive tokens and time. The optimal values varied by domain (security and logic agents required higher budgets than style and testing agents). Prompt brittleness was a recurring issue: sub-agents occasionally ignored structured output constraints and produced free-form text instead of valid JSON. This motivated the multi-stage parsing pipeline with progressive fallbacks, but even with these measures, some sub-agent runs produced no usable output and relied on the fallback path or the single-agent fallback.

Sandbox resource limits became relevant for larger repositories. The standard 8 GB memory allocation was sufficient for the evaluation repository but may be insufficient for monorepos with hundreds of thousands of source files. JSON parsing reliability from LLM outputs required the `fixJson` utility that handles trailing commas, single-quoted property names, unquoted keys, and line comments within JSON structures, but certain malformed outputs remain unrecoverable. The orchestrator occasionally introduced hallucinated findings or dropped valid ones during the LLM-based merge;

the deterministic fallback and confidence thresholds mitigate this but do not eliminate it.

4.8. Conclusion

This chapter presented the design, implementation, and evaluation of Falcon, demonstrating the feasibility of integrating AI-assisted pre-merge code review and post-deployment remediation within a unified platform. The proposed architecture combines shared infrastructure components with agent-based workflows to support both structured code analysis and autonomous runtime issue resolution. The implementation and evaluation results validate the system's design choices and demonstrate the potential of connecting review and remediation processes to provide more continuous and context-aware software quality assurance throughout the development lifecycle.

Conclusion

The rapid adoption of AI-assisted software development has increased code production and accelerated delivery cycles, creating new challenges for code review, quality assurance, and runtime reliability. Existing solutions typically address isolated stages of the software lifecycle, providing either pull request review assistance or production monitoring without establishing a continuous connection between the two.

To address this limitation, this work introduced Falcon, an AI-driven observability platform that integrates pre-merge code review with post-deployment remediation within a unified workflow. The proposed system combines a codebase graph engine, sandboxed execution environment, GitHub integration, and agent-based reasoning to support both structured pull request analysis and automated investigation of runtime failures. The evaluation demonstrated the feasibility of this approach, showing that Falcon can identify defects missed during human review and assist in the remediation of production issues through automatically generated, reviewable pull requests.

The results highlight the potential of combining repository-level code understanding with runtime observability to provide more continuous and context-aware software quality assurance. While the current implementation is limited to TypeScript repositories and relies on human validation before deployment, it establishes a practical foundation for integrating AI agents into modern development workflows.

Future work may focus on extending support to additional programming languages, improving the accuracy and robustness of agent reasoning, conducting larger-scale evaluations across diverse repositories, and incorporating more advanced verification mechanisms for automated remediation. These improvements would further strengthen Falcon's ability to support reliable, scalable, and intelligent software development practices.

In conclusion, this work highlights the potential of artificial intelligence to transform software quality assurance by bridging the gap between code review and runtime observability. Falcon demonstrates how intelligent agents can assist developers throughout the software lifecycle, helping teams identify defects earlier, respond to production incidents more effectively, and maintain higher levels of software reliability. We hope that this work contributes to the advancement of AI-assisted software engineering and serves as a foundation for future systems that combine human exper-

tise with autonomous reasoning to build more dependable, resilient, and trustworthy software systems.

References

[1] K. Tantithamthavorn et al., ``Rovodev code reviewer: A large-scale online evaluation of llm-based code review automation at atlassian,'' in *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*, ACM, 2026. [Online]. Available: https://arxiv.org/abs/2601.01129

[2] M. Kilic. ``Chronological history of version control systems,'' Accessed: May 31, 2025. [Online]. Available: https://muhammedkilic.medium.com/chronological-history-of-version-control-systems-9c648ed7b90c

[3] GeeksforGeeks. ``Evolution of software development: History, phases and future trends,'' Accessed: May 31, 2025. [Online]. Available: https://www.geeksforgeeks.org/software-engineering/evolution-of-software-development-history-phases-and-future-trends/

[4] Wikipedia contributors. ``History of software configuration management,'' Accessed: May 31, 2025. [Online]. Available: https://en.wikipedia.org/wiki/History_of_software_configuration_management

[5] Software Engineering Stack Exchange contributors. ``How did version control work on microcomputers of the day in the 80s and 90s?'' Accessed: May 31, 2025. [Online]. Available: https://softwareengineering.stackexchange.com/questions/310071/how-did-version-control-work-on-microcomputers-of-the-day-in-the-80s-and-90s

[6] Red-Gate Blog. ``A brief history of version control,'' Accessed: May 31, 2025. [Online]. Available: https://www.red-gate.com/blog/history-of-version-control/

[7] N. Constantino et al., ``Understanding collaborative software development,'' in *International Conference on Global Software Engineering*, CMU/ACM, 2020. [Online]. Available: https://www.cs.cmu.edu/~ckaestne/pdf/icgse20.pdf

[8] R. Yasrab, ``Challenges and issues in collaborative software developments,'' *arXiv preprint arXiv:1904.00721*, 2019. [Online]. Available: https://arxiv.org/pdf/1904.00721

[9] KMS Technology. ``7 key phases of the agile software development life cycle,'' Accessed: May 31, 2025. [Online]. Available: https://kms-technology.com/blog/key-phases-of-agile-software-development-life-cycle/

[10] Mendix. ``The 5 stages of the agile software development lifecycle,'' Accessed: May 31, 2025. [Online]. Available: https://www.mendix.com/blog/agile-software-development-lifecycle-stages/

[11] D. Strode et al., ``A teamwork effectiveness model for agile software development,'' *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–35, 2022. [Online]. Available: https://link.springer.com/article/10.1007/s10664-021-10115-0

[12] GitKraken. ``Version control for teams,'' Accessed: May 31, 2025. [Online]. Available: https://www.gitkraken.com/blog/version-control-for-teams

[13] BMC Blog. ``Github, gitlab, bitbucket & azure devops,'' Accessed: May 31, 2025. [Online]. Available: https://www.bmc.com/blogs/github-vs-gitlab-vs-bitbucket/

[14] rdnlsmith. ``A brief history of the pull request,'' Accessed: May 31, 2025. [Online]. Available: https://rdnlsmith.com/posts/2023/004/pull-request-origins/

[15] GitHub Docs. ``About pull requests,'' Accessed: May 31, 2025. [Online]. Available: https://docs.github.com/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests

[16] Rewind Blog. ``Github vs. bitbucket vs. gitlab,'' Accessed: May 31, 2025. [Online]. Available: https://rewind.com/blog/github-vs-bitbucket-vs-gitlab-comparison/

[17] LinearB Blog. ``What is a pull request & why is it important,'' Accessed: May 31, 2025. [Online]. Available: https://linearb.io/blog/what-is-a-pull-request

[18] GitHub Blog. ``Demonstrating end-to-end traceability with pull requests,'' Accessed: May 31, 2025. [Online]. Available: https://github.blog/enterprise-software/governance-and-compliance/demonstrating-end-to-end-traceability-with-pull-requests/

[19] GitHub Blog. ``Research: Quantifying github copilot's impact on developer productivity and happiness,'' Accessed: May 31, 2025. [Online]. Available: https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/

[20] S. Peng et al., ``The impact of ai on developer productivity: Evidence from github copilot,'' *arXiv preprint arXiv:2302.06590*, 2023. [Online]. Available: https://arxiv.org/abs/2302.06590

[21] Optimus Information. ``Unlocking developer productivity with github copilot: The future of ai-assisted coding,'' Accessed: May 31, 2025. [Online]. Available: https://www.optimusinfo.com/unlocking-developer-productivity-with-github-copilot-the-future-of-ai-assisted-coding/

[22] Nexos.ai. ``Best llms for coding and software development in 2026,'' Accessed: May 31, 2025. [Online]. Available: https://nexos.ai/blog/best-llm-for-coding/

[23] Milestone.ai. ``Integrating llms in software engineering teams,'' Accessed: May 31, 2025. [Online]. Available: https://mstone.ai/blog/practical-llm-approaches-engineering/

[24] LogRocket Blog. ``Why ai coding tools shift the bottleneck to review,'' Accessed: May 31, 2025. [Online]. Available: https://blog.logrocket.com/ai-coding-tools-shift-bottleneck-to-review/

[25] HDWebSoft. ``Ai augmented software development guide,'' Accessed: May 31, 2025. [Online]. Available: https://www.hdwebsoft.com/blog/ai-augmented-software-development

[26] Splunk. ``Ai-augmented software engineering,'' Accessed: May 31, 2025. [Online]. Available: https://www.splunk.com/en_us/blog/learn/ai-augmented-software-engineering.html

[27] The Virtual Forge. ``Why ai code needs human review: How to validate and secure ai-powered development,'' Accessed: May 31, 2025. [Online]. Available: https://www.thevirtualforge.com/company/blog/ai-generated-code-needs-human-oversight-how-to-validate-and-secure-ai-powered-development

[28] Graphite. ``Why ai will never replace human code review,'' Accessed: May 31, 2025. [Online]. Available: https://graphite.com/blog/ai-wont-replace-human-code-review

[29] GitHub Staff. ``Octoverse: A new developer joins github every second as ai leads typescript to #1,'' Accessed: May 31, 2026. [Online]. Available: https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/

[30] JetBrains. ``The state of ci/cd in 2025: Key insights from the latest jetbrains survey,'' JetBrains TeamCity Blog, Accessed: May 15, 2026. [Online]. Available: https://blog.jetbrains.com/teamcity/2025/10/the-state-of-cicd/

[31] Stack Overflow. ``2025 developer survey: Ai section,'' Accessed: May 31, 2026. [Online]. Available: https://survey.stackoverflow.co/2025/ai/

[32] GitClear. ``Ai usage impact: Correlation stats overview,'' Accessed: May 31, 2026. [Online]. Available: https://www.gitclear.com/help/ai_usage_impact_correlation_stats_overview

[33] CodeRabbit. ``Why 2025 was the year the internet kept breaking: Studies show incidents are increasing,'' CodeRabbit, Accessed: May 15, 2026. [Online]. Available: https://www.coderabbit.ai/blog/why-2025-was-the-year-the-internet-kept-breaking-studies-show-increased-incidents-due-to-ai

[34] Cole Stryker. ``What are large language models (llms)?'' IBM Think, Accessed: May 3, 2026. [Online]. Available: https://www.ibm.com/think/topics/large-language-models

[35] W. X. Zhao et al., ``A survey of large language models,'' *arXiv preprint arXiv:2303.18223*, 2023. [Online]. Available: https://arxiv.org/abs/2303.18223

[36] A. Vaswani et al., ``Attention is all you need,'' in *Advances in Neural Information Processing Systems*, 2017.

[37] Anna Gutowska. ``What are ai agents?'' IBM Think, Accessed: May 3, 2026. [Online]. Available: https://www.ibm.com/think/topics/ai-agents

[38] Cole Stryker. ``What is agentic ai?'' IBM Think, Accessed: May 3, 2026. [Online]. Available: https://www.ibm.com/think/topics/agentic-ai

[39] S. Yao, D. Yu, J. Zhao, I. Shafran, K. Narasimhan, and Y. Cao, ``React: Synergizing reasoning and acting in language models,'' in *International Conference on Learning Representations*, 2023.

[40] A. Bacchelli and C. Bird, ``Expectations, outcomes, and challenges of modern code review,'' in *Proceedings of the 2013 International Conference on Software Engineering*, IEEE, 2013, pp. 712–721.

[41] Greptile. ``Greptile: Ai code review agent,'' Accessed: May 31, 2025. [Online]. Available: https://www.greptile.com/docs/introduction

[42] Greptile. ``How greptile works: Graph-based codebase context,'' Accessed: May 31, 2026. [Online]. Available: https://www.greptile.com/docs/how-greptile-works/graph-based-codebase-context

[43] CodeRabbit. ``Coderabbit: Ai-powered code review,'' Accessed: May 31, 2025. [Online]. Available: https://docs.coderabbit.ai/guides/code-review-overview

[44] A. Deodhar. ``How ai code reviews actually work: A closer look at coderabbit,'' Accessed: May 31, 2026. [Online]. Available: https://aditideodhar.medium.com/how-ai-code-reviews-actually-work-a-closer-look-at-coderabbit-857ba8acd060

[45] Vercel. ``Vercel ai agent: Automated code review and fixes,'' Accessed: May 31, 2025. [Online]. Available: https://vercel.com/docs/agent/pr-review

[46] Cubic AI. ``Cubic: Ai code reviewer for pull requests,'' Accessed: May 31, 2025. [Online]. Available: https://docs.cubic.dev/ai-review/introduction

[47] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, ``Towards automating code review activities,'' *Proceedings of the 43rd International Conference on Software Engineering*, pp. 163–174, 2021.

[48] X. She et al., ``Pitfalls in language models for code intelligence: A taxonomy and survey,'' *ACM Computing Surveys*, 2023.

Chapter A Appendix: List of Acronyms

AI Artificial Intelligence

API Application Programming Interface

AST Abstract Syntax Tree

CI/CD Continuous Integration / Continuous Deployment

CLI Command-Line Interface

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

JSON JavaScript Object Notation

LLM Large Language Model

ML Machine Learning

NLP Natural Language Processing

OAuth Open Authorization

OWASP Open Web Application Security Project

PR Pull Request

ReAct Reasoning and Acting

ReWOO Reasoning Without Observation

RLHF Reinforcement Learning from Human Feedback

SDK Software Development Kit

UI User Interface