TUNISIAN REPUBLIC

Ministry of Higher Education and Scientific Research

University of Carthage

National Institute of Applied Sciences and Technology

Graduation Project

In order to obtain the

National Engineering Diploma

Specialty : **Software Engineering**

Enforcing Best Practices with LLM-IDE Integration

Presented by

Arij KOUKI

Hosted by

Google

INSAT Supervisor	:	**Ms. YOUSSEF Rabaa**
Company Supervisor	:	**Ms. LOPEZ Irene**

Presented on : –/–/**2025**

JURY

M. President	FLEN	(President)
Ms. Reviewer	FLENA	(Reviewer)

Academic Year : 2024/2025

TUNISIAN REPUBLIC

Ministry of Higher Education and Scientific Research

University of Carthage

National Institute of Applied Sciences and Technology

Graduation Project

In order to obtain the

National Engineering Diploma

Specialty : **Software Engineering**

Enforcing Best Practices with LLM-IDE Integration

Presented by

Arij KOUKI

Hosted by

Company Supervisor	**University Supervisor**

Academic Year : 2024/2025

Acknowledgements

I wish to extend my deepest gratitude and appreciation to everyone who has contributed significantly to the successful completion of this project.

My sincere thanks go to Irene Lopez and Andrew Xue, my host and co-host at Google Zurich, for their invaluable guidance, support, and trust throughout my internship. Their mentorship and kindness made this experience not only enriching but truly transformative.

Thank you to the YouTube Developer Infrastructure team for their warm welcome, collaborative spirit, and continuous encouragement, and to my mentor Veronica Radu, whose insights and support greatly contributed to my personal and professional growth.

I also wish to thank Mrs. Rabaa Youssef, my academic supervisor, for accompanying me in this final academic milestone.

To the distinguished members of the jury, I am grateful for your time and consideration in reviewing my work. I hope this report lives up to the standards expected of a graduation project.

To the professors of the National Institute of Applied Sciences and Technology, thank you for playing a vital role in shaping my academic and professional foundation.

And to my family and friends, your unwavering belief in me has carried me through the most demanding moments. Thank you for your love, patience, and presence.

Finally, a quiet note of gratitude to myself for the perseverance and dedication that made this journey possible.

Table of Contents

List of Figures

List of Tables

Résumé

Ce projet a été réalisé chez Google Zurich dans le cadre d'un Diplôme National d'Ingénieur en Génie Logiciel. Il explore l'intégration de l'intelligence artificielle générative dans le processus de développement logiciel en incorporant un agent basé sur un Large Language Model (LLM) au sein d'un Environnement de Développement Intégré (IDE) interne.

Cet agent effectue une analyse approfondie du code afin de détecter des violations complexes ou subjectives que les outils d'analyse statique traditionnels peuvent négliger. En générant des explications claires et compréhensibles ainsi que des suggestions concrètes, il aide les développeurs, notamment ceux contribuant à YouTube, à maintenir une haute qualité de code et à respecter les bonnes pratiques. Intégré de manière transparente au sein du flux de travail via une extension de l'IDE, l'agent améliore la productivité et contribue à la réduction de la dette technique sans perturber l'expérience de développement.

Ce travail met en évidence le potentiel des outils assistés par l'IA pour transformer l'expérience des développeurs et ouvre des perspectives pour l'avenir des environnements de développement intelligents.

Mots-clés : Intelligence Artificielle Générative, Génie Logiciel, Intégration IDE, Qualité du Code, Productivité des Développeurs

Abstract

This project was carried out at Google Zurich as part of a National Engineering Diploma in Software Engineering. It investigates the integration of generative artificial intelligence into the software development process by embedding a Large Language Model (LLM)-powered agent within an internal Integrated Development Environment (IDE).

The agent performs in-depth code analysis to detect complex or subjective violations that traditional static analysis tools may overlook. By generating clear, human-readable explanations and actionable suggestions, it supports developers, particularly those contributing to YouTube, in maintaining high code quality and adhering to best practices. Seamlessly integrated into the development workflow through an IDE extension, the agent enhances productivity and helps reduce technical debt without disrupting the coding experience.

This work demonstrates the potential of AI-assisted tooling to transform the developer experience and raises broader implications for the future of intelligent software engineering environments.

Keywords: Generative AI, Software Engineering, IDE Integration, Code Quality, Developer Productivity

General Introduction

The software engineering industry is experiencing a major shift driven by the rapid evolution of artificial intelligence and the growing demand for scalable, high-quality code development practices. As development teams grow and systems become more complex, ensuring consistent code quality and adherence to best practices presents an ongoing challenge, especially in large organizations managing massive codebases across distributed teams.

Traditional static analysis tools and linters, while helpful, often fall short when it comes to identifying nuanced or subjective coding issues that depend on context or internal guidelines. In fast-paced development environments, engineers need intelligent, responsive tools that go beyond rule-based checks to provide deeper insights and real-time guidance, without adding friction to their daily workflows.

This graduation project explores the integration of generative artificial intelligence into modern Integrated Development Environments (IDEs) to support software engineers in their day-to-day coding activities. The research investigates how Large Language Models (LLMs) can be leveraged to perform in-depth code analysis, detect complex violations, and offer clear, contextual suggestions for improvement. By embedding intelligent assistance directly within the development workflow, this work aims to enhance code quality, reduce technical debt, and support developer productivity at scale.

This report begins by establishing the theoretical foundation and examining the current state of AI-assisted development tools. It then presents the design and implementation of an intelligent code analysis system, followed by an evaluation of its effectiveness in real-world development scenarios. The work contributes to the broader understanding of how artificial intelligence can transform software engineering practices and improve developer experience.

Part I

Foundation and Context

Chapter I

Project Overview

Summary

Introduction

This opening chapter establishes the foundation of the graduation project by introducing the host company and defining the project's scope. We examine the organizational context, outline the main objectives and challenges, and present the methodological framework that guides the development process.

1 Host Company: Google

1.1 Presentation

Founded in 1998, Google LLC is a global leader in technology and innovation. As a subsidiary of Alphabet Inc., Google's mission is to organize the world's information and make it universally accessible and useful. Guided by values such as innovation, accessibility, sustainability, and user trust, Google has established itself as one of the most influential companies shaping the digital era. Its culture emphasizes collaboration, diversity, inclusion, and impact-driven engineering, enabling continuous leadership in research and product development.

Figure I.1 – Google Logo

1.2 Products and services

Google offers a broad ecosystem of products and services that touch nearly every aspect of digital life. Among its flagship consumer products are Google Search, Maps, Gmail, Chrome, and the Android operating system, serving billions of users daily.

Beyond consumer services, Google develops enterprise and cloud-based solutions such as Google Cloud Platform and Google Workspace, as well as advanced AI systems like Vertex AI. The company also invests in hardware, including Pixel devices, Nest smart home products, and ChromeOS.

These products reflect Google's commitment to connecting people, improving productivity, and driving digital transformation worldwide.

Figure I.2 – Overview of some Google products

1.3 Focus Area

1.3.1 YouTube

Acquired by Google in 2006, YouTube has become the world's leading video-sharing platform, serving more than two billion logged-in users monthly. It empowers individuals to create, share, and discover video content globally while sustaining a vibrant creator economy. From a technical perspective, YouTube integrates video processing, recommendation systems, live streaming, advertising, and trust and safety to deliver a seamless experience across devices.

Figure I.3 – YouTube Logo

1.3.2 YouTube Developer Infrastructure Team

Within YouTube's engineering organization, the Developer Infrastructure (Dev Infra) team supports thousands of engineers building the platform. The team develops tooling, automation, and guidelines that improve efficiency, reliability, and consistency in software development. By maintaining developer velocity and quality at scale, the Dev Infra team contributes directly to YouTube's ability to innovate and grow.

2 Project Overview

2.1 Project Context

This project was developed within the scope of a development infrastructure team dedicated to supporting developers by providing tools and extensions that enhance their daily workflows. As part of this mission, the team is exploring how artificial intelligence can be leveraged to assist developers in maintaining code quality and adhering to best practices. The goal is to investigate how large language models (LLMs) can complement traditional approaches by offering more intelligent and context-aware guidance directly within the IDE.

In parallel, this work also constitutes the mandatory fifth-year final project required for obtaining the software engineering degree at the National Institute of Applied Science and Technology, providing both academic and practical significance.

2.2 Problem Statement

In large-scale software development environments, maintaining uniform adherence to **internal framework–specific guidelines** across multiple teams is crucial for ensuring code quality, consistency, and long-term maintainability. While modern development environments provide assistance for general programming practices or widely used frameworks, they lack intelligent support for the nuanced, evolving rules of internal frameworks. As a result, developers often receive feedback on internal best practices only during code reviews, after significant effort has already been invested. This delayed feedback cycle leads to inefficiencies such as rework, slower iterations, and frustration among teams who must refactor code that was previously considered complete. The absence of real-time, context-aware guidance tailored to internal frameworks leaves developers navigating complex design decisions without adequate support, leading to technical debt, inconsistent quality, and higher onboarding complexity. Addressing this gap requires solutions that proactively enforce internal framework best practices during the coding phase, providing timely and context-specific feedback directly within the IDE.

2.3 Proposed Solution

This project introduces an **AI-assisted feedback system integrated directly into the coding workflow**. The solution is designed to address the challenges outlined above through three key capabilities:

- **Shift-Left Feedback:** Provide developers with earlier, context-aware guidance during the coding phase, ensuring that issues are detected and addressed well before code reviews.
- **Framework-Specific Best Practice Enforcement:** Surface adherence to internal framework guidelines early in the development process, going beyond syntax and correctness.
- **Reduced Review Burden:** Shift part of the best practice enforcement from manual reviews to the authoring stage, allowing reviews to focus on higher-level insights.

By integrating intelligent, framework-aware feedback directly into the coding workflow, this solution aims to minimize rework, improve adherence to internal standards, and accelerate development velocity.

3 Work Methodology

3.1 Agile Development Approach

Agile software development, as defined by Beck et al. [1], emphasizes ”individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan.” This methodology was adopted to support iterative development and maintain flexibility in responding to evolving requirements. According to Martin [2], agile practices enable continuous integration of feedback, ensuring that each increment of work aligns with both technical goals and the broader product vision. Testing, validation, and code reviews were incorporated throughout the process to maintain high quality, while frequent collaboration provided clarity and shared ownership of outcomes. Agile principles complemented the focus on engineering excellence, including rigorous design reviews, thorough testing, robust code reviews, and DevOps-enabled automation.

3.2 Kanban Workflow

Kanban, as described by Anderson [3], is ”a method for managing knowledge work with an emphasis on just-in-time delivery while not overloading the team members.” The dynamic workload of the development infrastructure team, including feature requests, bug fixes, and maintenance tasks, was managed using this Kanban workflow. According to Kniberg and Skarin [4], Kanban enables teams to visualize tasks and limit work in progress, preventing bottlenecks and allowing quick focus shifts to urgent issues when necessary. Work was structured into stages to maintain clear coordination while allowing the flexibility to adapt priorities as requirements evolved.

The Kanban workflow included the following stages:

- **Backlog:** Prioritized collection of feature requests, enhancements, and bug fixes.
- **Research & Design:** Assessment of technical feasibility and preparation of design specifications.
- **Development:** Implementation and integration of features into the system.
- **Review & Testing:** Code review, unit tests, and integration tests to ensure quality and correctness.
- **Deployment:** Release of validated features to developer environments.

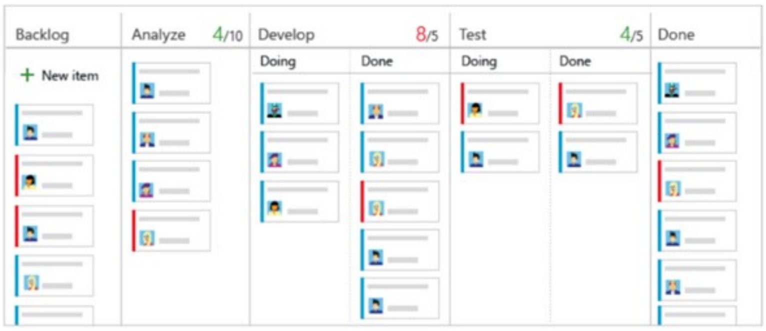


Figure I.4 – Kanban Workflow

3.3 Development Process

The project followed an iterative engineering cycle designed to balance thorough planning with incremental delivery, as illustrated in Figure I.5. This cycle guided the work through structured stages, supported by dedicated tools and regular collaboration:

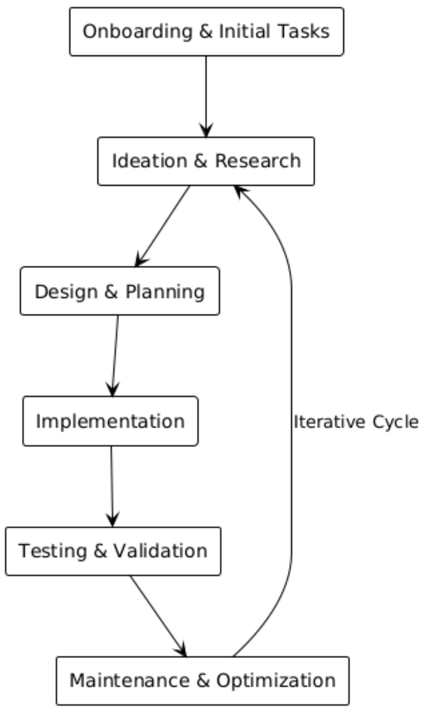


Figure I.5 – Project Development Cycle

The main stages, as depicted in the figure, included:

- **Onboarding and Initial Tasks:** The project began with a structured onboarding phase, where familiarization with internal tools, repositories, and coding standards was combined with the resolution of assigned bugs. This phase ensured a smooth transition into the team's workflow and provided early practical experience.

- **Ideation and Research:** Following onboarding, the project entered an exploration phase to clarify objectives, gather requirements, and investigate potential solution directions. Independent research was complemented by collaborative discussions to assess feasibility and align priorities.
- **Design and Planning:** A detailed design document was authored to present technical choices, architectural considerations, and the proposed workflow. The document underwent iterative review by engineers, and the final approved plan was transferred to the internal task management system for structured tracking and prioritization.
- **Implementation:** Development was performed in small, reviewable increments using the internal development environment within the company-wide repository. Each change was submitted with unit tests and validated through manual and AI-assisted code reviews.
- **Testing and Validation:** Functionality and reliability were verified continuously. Automated unit tests ensured correctness at the component level, while integration reviews and structured evaluations validated the behavior within the larger system.
- **Maintenance and Optimization:** Refactoring, bug fixes, and updates were performed throughout development, particularly as dependencies evolved or methods became deprecated. This ensured that the solution remained consistent, maintainable, and aligned with evolving standards.

Collaboration was supported through a structured communication rhythm, combining regular syncs with the host and co-host, weekly team meetings, and occasional cross-team discussions. This cadence provided timely feedback, clear guidance, and alignment on shared dependencies throughout the iterative cycle.

3.4 Project Timeline

The project spanned four months (May 5 – September 5, 2025). Work was scheduled based on business priorities and technical dependencies. Early weeks focused on research and design, followed by implementation, testing, and iterative refinement.

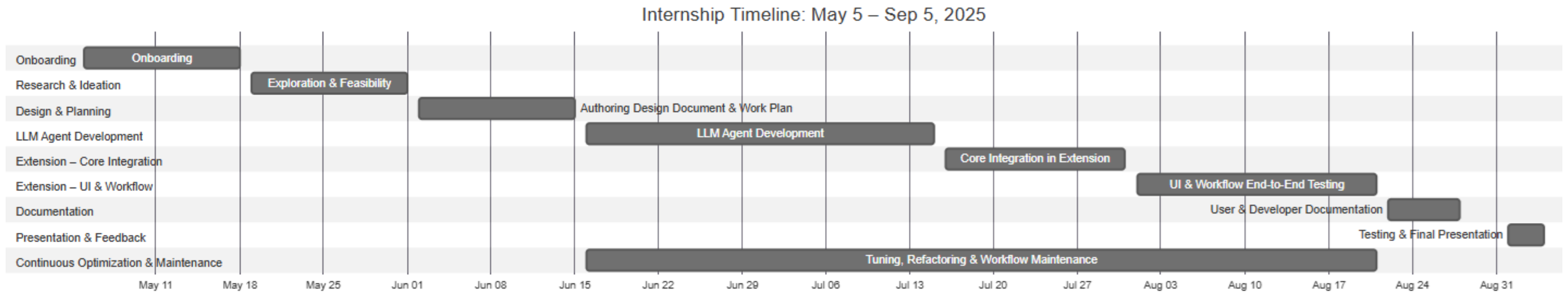


Figure I.6 – Project Timeline - Detailed Schedule

Conclusion

In summary, this chapter outlined the project's context by presenting the host company, defining the problem, and clarifying the main objectives and challenges. It also described the methodology chosen to guide the work, which provides the basis for the technical developments detailed in the following chapters.

Part II

Theoretical Framework

Chapter II

Business Understanding and State of the Art

Summary

Introduction

This chapter establishes the theoretical and contextual foundation of the project. It begins with an overview of the Software Development Life Cycle (SDLC) and the emerging role of AI technologies such as Large Language Models (LLMs), Generative AI, and AI agents in modern software engineering. It then presents the state of the art, reviewing existing solutions that support code quality and best practices. Finally, it defines the project requirements, both functional and non-functional, showing how this work addresses gaps by integrating AI-driven assistance into key SDLC phases.

1 Business and Reasoning

1.1 Software Development Life Cycle (SDLC)

The Software Development Life Cycle (SDLC) provides a structured framework for planning, creating, testing, deploying, and maintaining software systems. As development teams grow and projects become more complex, maintaining code quality and consistency across all phases becomes increasingly challenging. As shown in Figure II.1, each phase presents unique challenges that can impact developer productivity and software reliability.

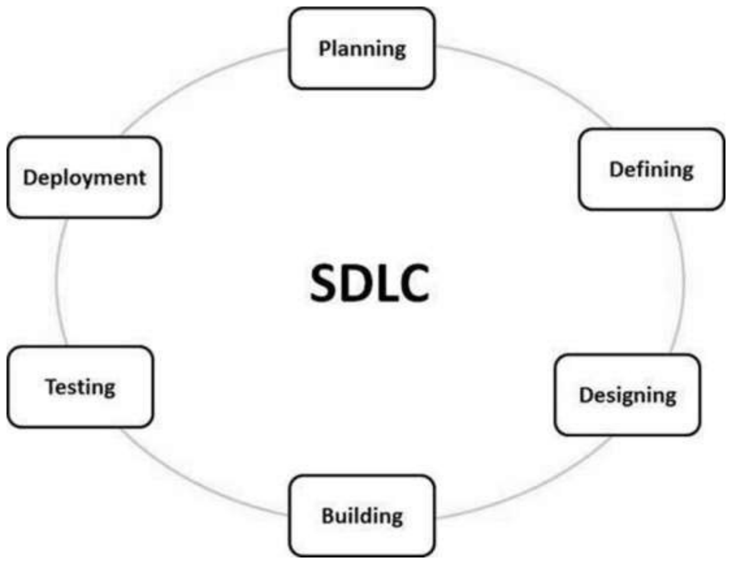


Figure II.1 – Software Development Life Cycle Overview

Key Development Challenges

Modern software development faces several critical challenges:

- **Scalability Issues:** Maintaining consistent coding standards and best practices becomes increasingly difficult as teams grow, particularly without automated assistance.
- **Quality Assurance Gaps:** Traditional QA relies heavily on human review, introducing delays and inconsistent feedback timing.
- **Technical Debt Accumulation:** Without timely guidance, developers may adopt patterns that violate best practices, leading to accumulating technical debt.
- **Knowledge Transfer Challenges:** New team members often learn framework-specific

conventions and best practices through trial and error, slowing onboarding and introducing variability.

Modern Development Approaches

Contemporary practices aim to mitigate these challenges:

- **Agile Methodologies:** Emphasize iterative development, continuous feedback, and rapid adaptation to changing requirements.
- **DevOps Integration:** Combines development and operations practices to enable continuous integration, automated testing, and real-time monitoring.
- **AI-Enhanced Development:** Emerging AI technologies provide intelligent assistance throughout the development process, from code generation to quality assurance.

These approaches lay the foundation for understanding how AI can address persistent challenges in code quality and consistency across modern software teams.

1.2 Artificial Intelligence in Software Development

Foundational AI Concepts

Artificial Intelligence (AI) refers to computational systems capable of performing tasks that typically require human intelligence, such as reasoning, learning, problem-solving, and decision-making. AI encompasses a broad spectrum of techniques and paradigms with distinct capabilities and applications. Figure II.2 provides a visual hierarchy of AI technologies.

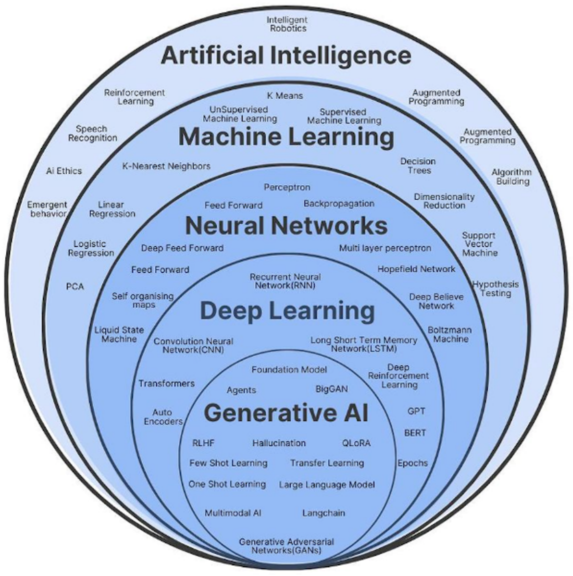


Figure II.2 – Hierarchy of AI Technologies

Key categories include:

- **Symbolic AI:** Rule-based systems for reasoning and knowledge representation.
- **Machine Learning (ML):** Algorithms that learn from data to improve task performance.
- **Deep Learning (DL):** Neural networks with multiple layers capable of modeling complex patterns.
- **Generative AI:** Systems that produce new content, such as text or code, by learning patterns from existing datasets.

The AI Revolution in Software Engineering

As software development becomes increasingly complex, traditional approaches struggle to maintain quality while meeting delivery deadlines. AI technologies offer transformative opportunities by embedding intelligent assistance directly into the development workflow.

Industry adoption illustrates this impact: surveys indicate that AI-generated code accounts for a significant portion of development output in major tech organizations [5, 6]. AI integration addresses critical business challenges, including maintaining code quality, reducing technical debt, and scaling development practices across growing teams. This shift defines the AI-Enhanced SDLC, a lifecycle where intelligent assistance is embedded throughout all phases.

Large Language Models (LLMs)

LLMs represent a breakthrough in generative AI, trained on extensive natural language and code corpora. Unlike traditional tools, they understand context, semantics, and intent, enabling complex reasoning across domains. Figure II.3 shows a chronological overview of LLM development from 2018–2024.

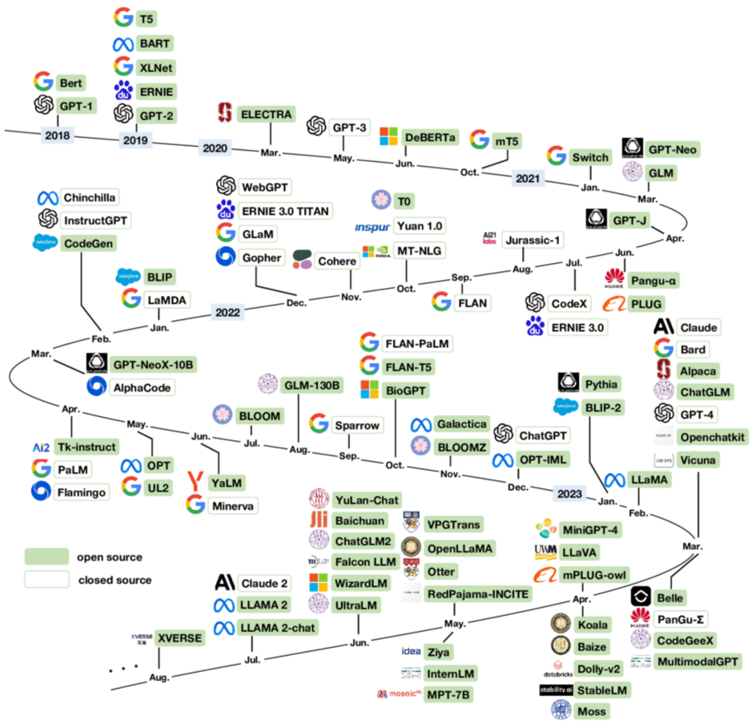


Figure II.3 – Chronological Overview of Large Language Models (LLMs)

LLM capabilities include:

- **Natural Language Understanding**
- **Pattern Recognition**
- **Content Generation**
- **Reasoning and Inference**

Applied to software development, these translate into:

- **Contextual Code Analysis**
- **Intelligent Code Generation**

- **Explanatory Documentation**
- **Semantic Standards Enforcement**

AI Agents

AI agents build on LLMs by autonomously reasoning, planning, and executing development tasks. They orchestrate LLM capabilities within workflows, providing seamless integration into IDEs, testing frameworks, and version control systems. Figure II.4 illustrates the agent architecture and orchestration workflow.

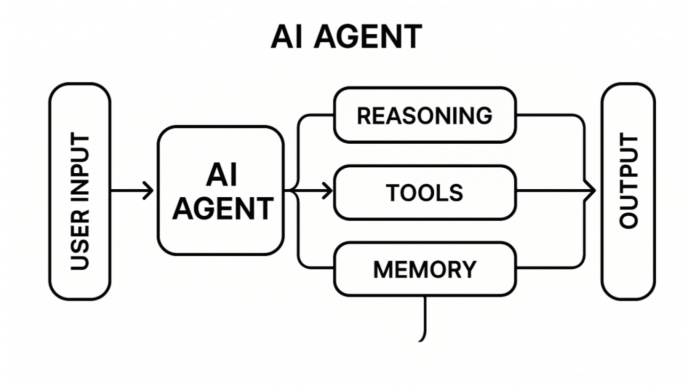


Figure II.4 – AI Agent Architecture and Orchestration Workflow

Core components include:

- Code Analysis Engine
- Contextual Reasoning
- Development Tool Integration
- Contextual Adaptation via system prompts
- Feedback and Explanation System

Balancing AI Capabilities and Practical Considerations

Capabilities:

- Contextual Understanding
- Automated Analysis

- Intelligent Recommendations
- Project-specific Adaptation

Limitations:

- Computational Cost
- Context Constraints
- Accuracy Considerations
- Integration Complexity
- Adaptation via prompts rather than continuous learning

AI-Enhanced SDLC

The combination of LLMs and AI agents enables proactive, intelligent assistance throughout the entire software development lifecycle. This AI-enhanced SDLC transforms traditional development practices by embedding guidance, analysis, and automated support at each phase.

Transformative Impact on Development Practices

- **From Reactive to Proactive:** Traditional workflows provide feedback mainly during code reviews or testing phases. AI-enhanced systems deliver guidance during coding, design, and planning, preventing issues before they become embedded in the codebase and reducing the need for extensive refactoring.
- **From Inconsistent to Scalable:** Human-based quality assurance can vary in expertise and availability. AI agents provide consistent, expert-level analysis across large teams and codebases, ensuring uniform adherence to best practices and internal framework conventions.
- **From Static to Adaptive:** Unlike rule-based tools, AI adapts to project-specific patterns, team preferences, and evolving best practices, offering context-aware suggestions tailored to the particular codebase.
- **From Isolated to Integrated:** Instead of treating quality assurance and guidance as separate phases, AI agents integrate intelligent support directly into developer workflows, bridging planning, implementation, testing, and maintenance.

AI Applications Across the SDLC AI technologies provide targeted assistance at each SDLC phase, addressing specific challenges and improving efficiency:

- **Requirements and Planning:** AI analyzes historical project data to estimate timelines, identify ambiguities in requirements, and suggest optimal resource allocation. Natural language processing helps translate business requirements into technical specifications with higher precision.
- **Design and Architecture:** AI assists architects by analyzing existing codebases, recommending architectural patterns, detecting design anti-patterns, and ensuring alignment with organizational standards. This reduces the likelihood of systemic flaws early in the project lifecycle.
- **Implementation:** During coding, AI agents provide context-aware code completions, enforce framework-specific best practices, detect potential bugs, and suggest refactoring opportunities. This reduces rework, accelerates development, and supports consistent code quality.
- **Testing and Quality Assurance:** AI generates comprehensive test cases, identifies edge cases that might be missed manually, prioritizes test execution based on risk, and evaluates code quality against defined standards. This enhances reliability and reduces technical debt accumulation.
- **Deployment and Maintenance:** AI monitors deployment health, predicts potential issues, and recommends optimizations based on performance metrics and usage patterns. Continuous insights help maintain system stability and inform future development iterations.

Overall, the AI-enhanced SDLC shifts the development workflow from reactive and fragmented practices toward a more proactive, adaptive, and integrated approach. By embedding intelligent assistance at every stage, it directly addresses the challenges of scaling teams, maintaining consistent quality, and supporting internal framework-specific best practices—core objectives of this project.

2 State of the Art and Existing Solutions

This section examines both market-available solutions (state of the art) and environment-specific approaches (existing solutions) to understand the current landscape of code quality and best practices enforcement.

2.1 State of the Art: Market Solutions

The software development market offers various AI-powered tools and platforms that aim to enhance code quality and developer productivity. These solutions represent the current state of the art in intelligent development assistance:

AI-Powered Code Analysis Tools Commercial and open-source solutions provide intelligent code analysis capabilities:

- **AI Code Assistants:** Tools like GitHub Copilot, Amazon CodeWhisperer, and Tabnine provide AI-powered code completion and generation, helping developers write code more efficiently.
- **AI-Powered IDEs:** Modern development environments like Cursor and Claude Code integrate AI assistance directly into the coding workflow, offering context-aware code generation, refactoring, and intelligent suggestions.
- **Static Analysis Platforms:** Solutions such as SonarQube, CodeClimate, and DeepCode offer automated code quality analysis with AI-enhanced pattern detection.
- **AI Code Review Tools:** Platforms like PullRequest.com and CodeRabbit provide AI-assisted code review capabilities, offering automated suggestions and quality assessments.

Market Solution Capabilities These market solutions typically offer:

- **General Code Analysis:** Broad pattern recognition and quality assessment across multiple programming languages and frameworks.
- **AI-Powered Suggestions:** Intelligent recommendations for code improvements, refactoring, and best practices.
- **Integration with Popular IDEs:** Seamless integration with widely-used development environments like VS Code, IntelliJ, and Eclipse.

2.2 Existing Solutions: Environment-Specific Approaches

Within our specific development environment, software engineers currently rely on a combination of modern AI-powered tools and traditional approaches to maintain code quality and enforce best practices:

Current Feedback Mechanisms The existing approaches in our environment can be categorized into several key mechanisms:

- **Code Reviews:** Human reviewers examine code for design quality, readability, maintainability, and adherence to standards. While this approach provides high-level, context-aware feedback, it often introduces delays and requires significant effort.
- **Presubmit Checks:** Automated scripts that validate code before submission, enforcing style guides, ensuring compilation, and verifying simple correctness and safety constraints. Although fast and reliable, they primarily focus on surface-level checks and do not consider design or contextual issues.
- **Coding Assistant:** Our internal IDE includes an integrated coding assistant that provides AI-powered code completion, generation, and basic suggestions to help developers write code more efficiently.
- **Linters:** Live IDE-integrated linters that catch style issues, deprecations, and basic code quality problems in real-time as developers write code. These tools provide immediate feedback on syntax, formatting, and some best practices.
- **Rule-Based Checks:** Systems that enforce coding conventions, naming schemes, and formatting standards. These tools provide consistency and objectivity but cannot reason about complex or context-dependent best practices.

Gap Analysis: Market vs. Environment Solutions While both market solutions and environment-specific approaches provide valuable capabilities, they leave significant gaps in addressing internal framework-specific best practices:

Market Solution Limitations:

- **Generic Analysis:** Market tools provide general code quality analysis but lack deep understanding of internal framework-specific conventions and patterns.
- **External Dependency:** Commercial solutions require external data sharing and may not align with internal security and privacy requirements.
- **Limited Customization:** Generic tools cannot be easily customized to enforce internal-specific best practices and architectural patterns.

Environment Solution Limitations: While our environment includes a coding assistant and live linters, there remains a significant gap in providing intelligent feedback specifically for internal framework best practices during the active coding phase. This timeline illustrates where current feedback mechanisms fit into the developer workflow:

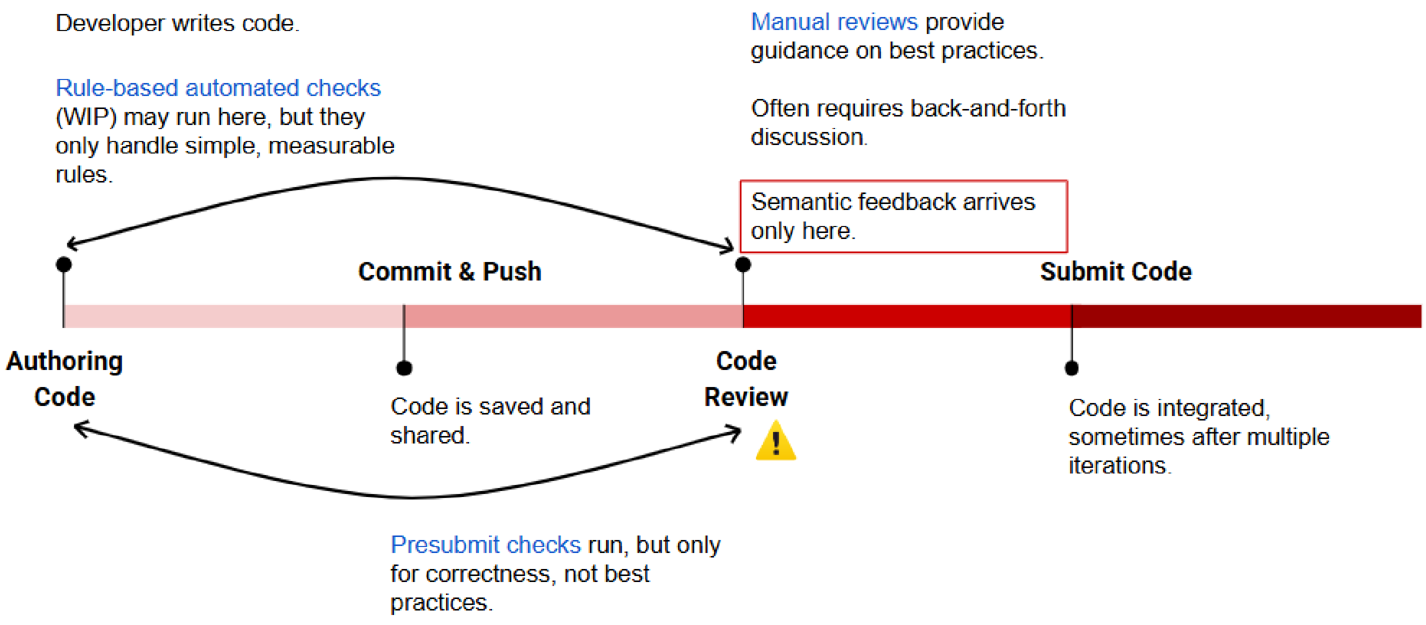

Developer writes code.
Rule-based automated checks (WIP) may run here, but they only handle simple, measurable rules.
Manual reviews provide guidance on best practices.
Often requires back-and-forth discussion.
Semantic feedback arrives only here.
Commit & Push
Submit Code
Authoring Code
Code is saved and shared.
Code Review
Code is integrated, sometimes after multiple iterations.
Presubmit checks run, but only for correctness, not best practices.

Figure II.5 – Current Developer Workflow Feedback Timeline

During the **Coding Phase**, developers get support from the Coding Assistant for code generation and completion, and from Linters for style issues and deprecations. However, these tools focus on general coding assistance and basic quality checks rather than internal framework-specific best practices.

The most insightful feedback on design and best practices typically comes during **Code Review**, but this happens after the initial development, making changes more disruptive.

Finally, **Presubmit Checks** before submission focus on correctness and safety, not on proactive best practice guidance.

This leaves a significant gap: there's no real-time, intelligent support for adhering to internal framework best practices while the developer is actively coding, which is the gap our project aims to fill.

Approach	**Strengths**	**Limitations**
Code Reviews	Context-aware, high-level insights	Feedback delayed, time-consuming
Presubmit Checks	Fast, automated validation	Limited scope, mostly syntax and safety
Coding Assistant	AI-powered code generation, completion	General assistance, not framework-specific
Linters	Real-time style and deprecation checks	Basic quality only, not best practices
Rule-Based Checks	Consistency, objectivity	Cannot handle complex or contextual practices

Tableau II.1 – Comparison of available software quality support solutions

Impact of Delayed Feedback Following from the previous analysis, these workflow issues create concrete impacts on development:

- **Technical Debt Accumulation:** Delayed feedback and limited enforcement of best practices lead to accumulating technical debt and inconsistencies in code quality over time.
- **Prolonged Review Cycles:** Review cycles take longer because developers must iterate multiple times to address issues discovered late in the process.
- **Developer Frustration:** The repetitive cycle of late-stage corrections contributes to developer frustration and reduced productivity.
- **Inconsistent Quality Standards:** Without real-time guidance, adherence to best practices varies significantly across team members and projects.
- **Increased Development Costs:** The cost of fixing issues increases exponentially the later they are discovered in the development process.

This analysis clearly demonstrates why we need a solution that brings intelligent best practices guidance earlier, directly into the authoring process, addressing the specific gap in internal framework best practices enforcement that current approaches cannot fill.

Opportunity for Framework-Specific AI Solutions The analysis reveals a clear opportunity for developing framework-specific AI solutions that combine the intelligence of market tools with the specificity required for YouTube framework best practices. While market solutions provide general AI capabilities and environment solutions offer domain knowledge, neither adequately addresses the need for real-time, intelligent feedback tailored to YouTube framework conventions.

This gap creates a compelling case for integrating LLM-powered assistance directly into the YouTube development workflow, providing intelligent, context-aware guidance that understands both general best practices and YouTube-specific patterns.

—

3 Project Requirements

To tackle the issue of delayed feedback and limited intelligent assistance, our solution integrates the power of large language models directly into the developer's workflow in the IDE.

The proposed system builds on gaps identified in existing solutions. It aims to provide real-time, AI-driven feedback directly in the developer workflow, while maintaining performance, scalability, and usability.

3.1 Functional Requirements

The system must:

- **Detect Framework Violations:** Identify violations of internal YouTube framework best practices and coding standards in real-time during development.
- **Provide Contextual Explanations:** Explain violations in clear, developer-friendly language with context-specific rationale that helps developers understand why certain patterns are problematic.
- **Generate Actionable Fixes:** Suggest specific, actionable fixes leveraging AI-generated solutions tailored to YouTube framework patterns and conventions.
- **Enable Developer Interaction:** Allow developers to accept, reject, or modify AI suggestions, providing full control over the implementation of recommended changes.
- **Maintain Contextual Relevance:** Ensure feedback appears in the appropriate location within the code and remains relevant and accurate as the developer continues coding.
- **Seamless IDE Integration:** Integrate seamlessly with the existing internal IDE, extension, and backend workflow without disrupting the developer's current development process.

3.2 Non-Functional Requirements

The system should also meet broader quality criteria:

- **Performance:** Provide fast, near real-time responses to avoid interrupting developer workflow.
- **Scalability:** Efficiently handle large codebases and multiple simultaneous users.
- **Maintainability:** Enable modular updates, addition of new AI models, or coding rules.
- **Reliability:** Ensure robustness in production environments with minimal downtime.
- **Security and Privacy:** Comply with organizational policies, ensuring safe handling of code and data.
- **Usability:** Deliver concise, context-aware, and minimally intrusive feedback.
- **Extensibility:** Easily add new rules, models, or integrations.

Requirement Type	**Description**
Functional	Framework violation detection, contextual explanations, actionable fixes, developer interaction, contextual relevance, seamless IDE integration
Non-Functional	Performance, scalability, maintainability, reliability, security, usability, extensibility

Tableau II.2 – Summary of project requirements

Rationale These requirements address limitations identified in existing solutions by embedding proactive, context-aware feedback directly in the coding workflow, specifically targeting YouTube framework best practices. This approach enhances developer productivity, reduces framework-specific errors, and supports consistent adherence to internal YouTube development standards.

Conclusion

This chapter established the business and theoretical foundation for integrating AI into YouTube framework development workflows. Beginning with an analysis of modern development challenges, it identified key issues in maintaining code quality and consistency across growing teams. The exploration of AI technologies—particularly Large Language Models and AI agents—revealed their potential for addressing these challenges through proactive, context-aware assistance.

The examination of both market solutions and environment-specific approaches highlighted significant limitations: market tools lack framework-specific knowledge, while traditional environment solutions provide delayed feedback and limited scope. There remains a critical gap in delivering real-time, intelligent assistance tailored to YouTube framework best practices.

The project requirements defined in this chapter focus on bridging this gap through real-time, AI-driven feedback that integrates seamlessly into the YouTube development workflow. The proposed solution combines the intelligence of modern AI technologies with the specificity required for YouTube framework conventions, setting the stage for the detailed system design presented in the following chapter.

Part III

System Design, Implementation and Evaluation

Chapter III

System Design and Architecture

Summary

Introduction

This chapter presents the system design and architecture of the LLM-powered best practices enforcement system. Building on the business understanding and requirements established in the previous chapter, this chapter details the technical design decisions, architectural patterns, and system components that enable real-time, intelligent feedback for YouTube framework development.

The design follows traditional software engineering principles while incorporating modern AI technologies. This chapter covers the overall system architecture, component design, data models, and integration patterns that form the foundation of the implemented solution.

1 System Architecture Overview

1.1 Use Case Analysis

Understanding the system's interactions with its users is fundamental to designing an effective solution. The use case analysis identifies the primary actors and their interactions with the LLM Best Practices Enforcement System, establishing the scope and boundaries of the system's functionality.

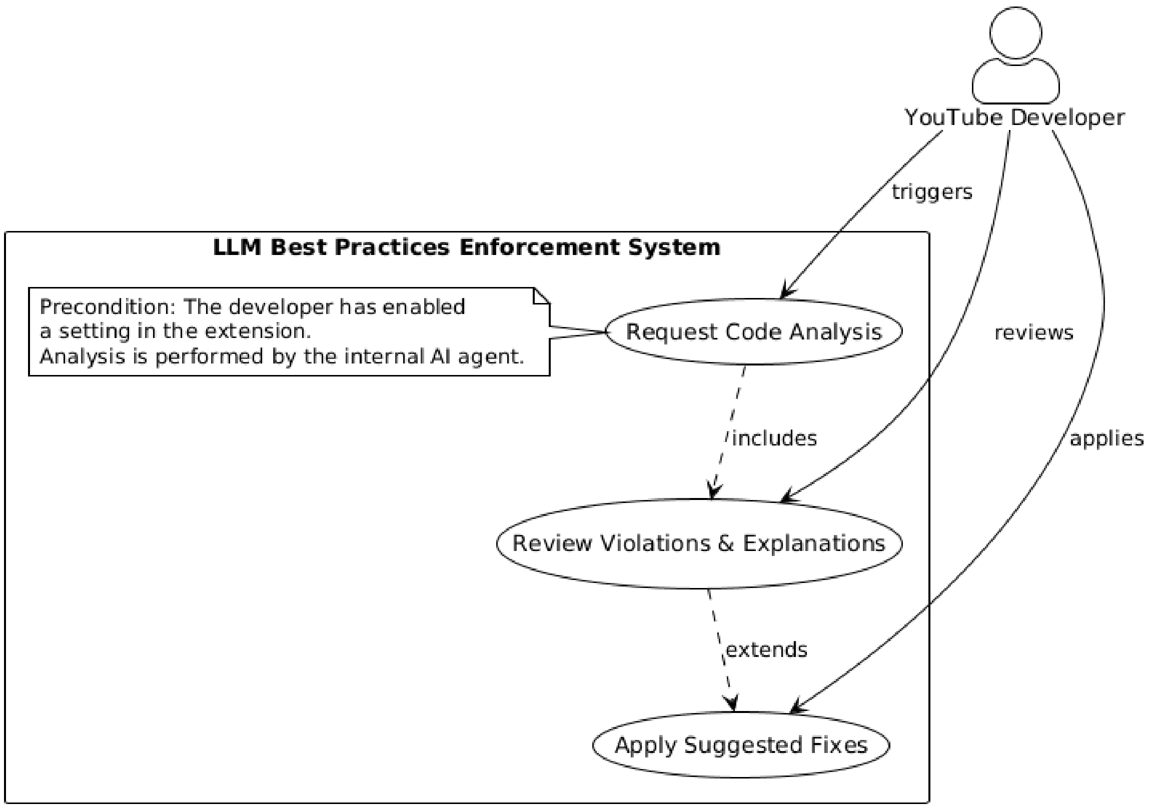

YouTube Developer
triggers
reviews
applies
LLM Best Practices Enforcement System
Precondition: The developer has enabled
a setting in the extension.
Analysis is performed by the internal AI agent.
Request Code Analysis
includes
Review Violations & Explanations
extends
Apply Suggested Fixes

Figure III.1 – System Use Case Diagram

This use case diagram illustrates the core functionality of the system from the perspective of YouTube developers. The primary actor is the YouTube Developer, who interacts with the system through three main use cases: requesting code analysis, reviewing violations and explanations, and optionally applying suggested fixes. The relationship between use cases reflects the natural workflow: analysis always includes reviewing results, while applying fixes is an optional extension that developers can choose based on their needs and preferences. This design ensures that developers maintain control over their workflow while providing comprehensive feedback when requested. The diagram focuses on the most important scenarios that occur during normal system usage, avoiding authentication and administrative scenarios that are less central to the user experience.

1.2 High-Level Architecture

The system architecture is designed to integrate seamlessly into the developer's existing workflow while providing intelligent, context-aware feedback. The architecture consists of two main components that work together to deliver real-time best practice enforcement:

- **IDE**: The developer's workspace containing the YouTube IDE Extension, which works with the currently open file being analyzed and annotates results directly in the editor.
- **AI Agent Framework**: The processing layer containing the LLM Best Practices Agent, which serves as the core AI processing engine that uses internal LLMs to analyze code and suggest best practices.

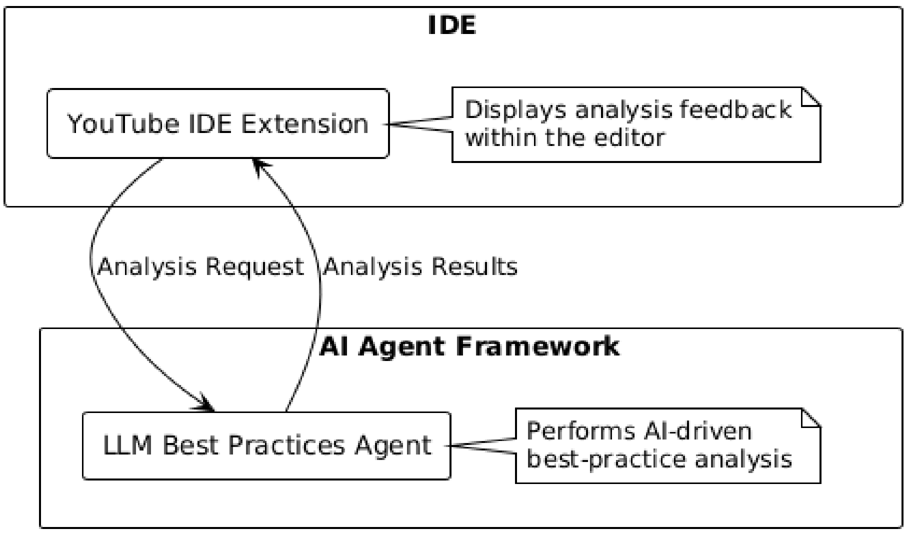


Figure III.2 – High-Level System Architecture

This architecture diagram illustrates the fundamental separation between the user-facing IDE extension and the AI processing backend. The YouTube IDE Extension operates within the developer's workspace, providing immediate access to analysis capabilities while maintaining the familiar development environment. The AI Agent Framework handles the computationally intensive analysis tasks, ensuring that the IDE remains responsive during processing. This separation enables independent scaling of AI capabilities without impacting the development environment's performance.

1.3 System Workflow

The system operates through a streamlined workflow that begins when a developer triggers analysis via the YouTube IDE Extension. The complete interaction flow is depicted in Figure III.3.

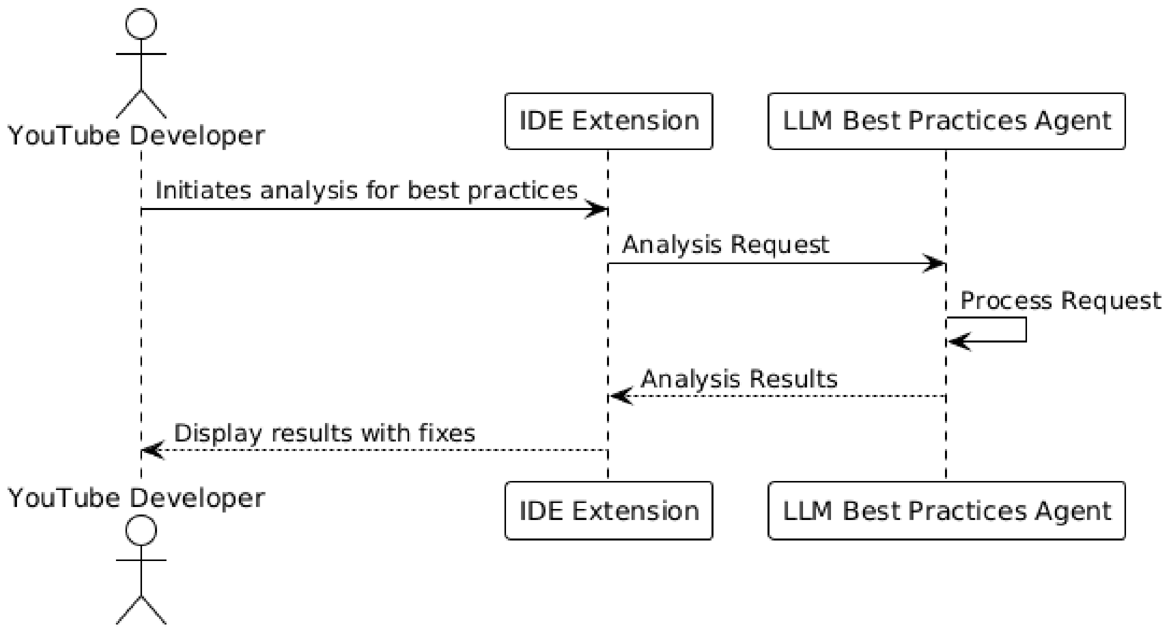


Figure III.3 – System Interaction Sequence Diagram

This sequence diagram captures the most important interaction scenario: a developer requesting analysis of their current file. The diagram shows the complete flow from user action to result presentation, demonstrating how the system maintains responsiveness by delegating heavy processing to the AI agent while providing immediate feedback through the IDE extension. This represents the primary use case that occurs most frequently in the system.

The sequence unfolds through the following interactions and responsibilities:

1. **Analysis initiation**: The YouTube developer triggers analysis of the currently open file.
2. **Request submission**: The YouTube IDE Extension submits an analysis request to the agent, identifying the target file.
3. **Processing**: The agent performs best-practices analysis and prepares the resulting findings.
4. **Result delivery**: The agent returns a structured response to the YouTube IDE Extension.

5. **Presentation**: The YouTube IDE Extension presents the best practices violations and suggested fixes within the editor.

This workflow emphasizes decoupling the IDE from heavy AI computation, ensuring that the development environment remains responsive while delegating intensive analysis to the specialized agent framework.

2 Components Design

2.1 LLM Best Practices Agent

The LLM Best Practices Agent serves as the core intelligence engine of the system, responsible for analyzing code, identifying violations of YouTube framework best practices, and providing actionable feedback to developers. This component represents the convergence of modern AI capabilities with domain-specific software engineering expertise.

2.1.1 Agent Architecture Choice

The agent is built using the *Executable Agent* architecture [7], provided by our internal AI platform. This architecture offers a structured approach to orchestrating AI-powered workflows, and was selected after careful evaluation of different agent paradigms, considering reliability, performance, and maintainability.

Executable Agent vs. ReAct Architecture To motivate the choice of *Executable Agent* architecture, we contrast it with the more widely known ReAct (Reasoning and Acting) pattern. In our system, the *Executable Agent* refers to a deterministic, tool-orchestrated workflow where control flow is defined in code rather than delegated to LLM reasoning.

- **ReAct Agent**: Uses a reasoning loop where the LLM decides what action to take next, executes it, observes the result, and continues reasoning. This creates a dynamic, LLM-driven execution flow.
- **Executable Agent**: Follows a predefined, deterministic execution flow where the agent orchestrates a sequence of tool calls in a structured manner, with the LLM used primarily for processing within each tool rather than for orchestration decisions.

Figure III.4 illustrates the fundamental difference in control flow between the two approaches.

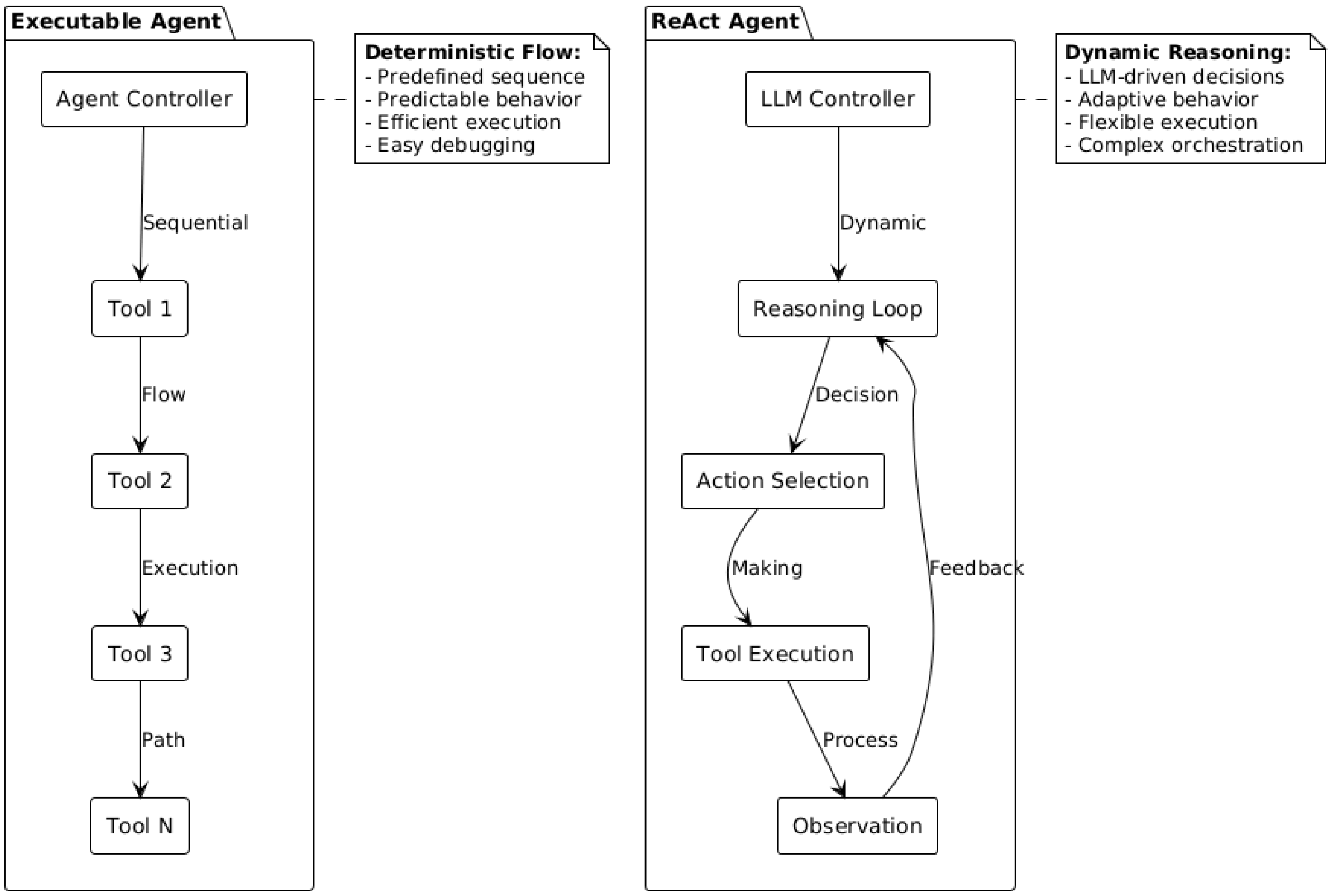


Figure III.4 – Comparison of Executable Agent vs. ReAct Agent Architectures

The Executable Agent architecture offers several key advantages for this use case:

- **Deterministic Execution**: Predefined flow ensures predictable behavior and simplifies debugging.
- **Tool Orchestration**: Clean framework for coordinating multiple specialized tools.
- **Error Handling**: Built-in mechanisms for handling failures and graceful degradation.
- **Performance**: Efficient execution with low response latency, since orchestration decisions are pre-programmed rather than deferred to open-ended LLM reasoning.
- **Cost Efficiency**: Reduced token consumption by constraining LLM calls to focused, well-scoped processing steps rather than repeated reasoning loops.
- **Maintainability**: Clear separation of concerns between tools and responsibilities.

2.1.2 Core Tools Architecture

To achieve this structured workflow, the agent incorporates five specialized tools, each handling a specific step of the best practices analysis pipeline.

Tool Responsibilities The agent's tool architecture follows a sequential workflow where each tool has a distinct responsibility:

- **File Reading Tool**: Responsible for retrieving the complete content of the file being analyzed, ensuring the agent has access to the full context of the code under examination.
- **Code Analysis Tool**: The core analysis engine that performs violation detection against YouTube framework best practices using semantic code analysis to understand intent and context.
- **Violation Explanation Tool**: Generates human-readable explanations for each identified violation, providing developers with clear understanding of why particular code patterns violate best practices.
- **Code Fix Tool**: Proposes actionable fix suggestions for identified violations, going beyond problem identification to provide concrete solutions that developers can implement.
- **Result Consolidation Tool**: Aggregates all analysis results into a structured output format, ensuring the response contains all necessary information for the IDE extension to display results effectively.

This tool-based architecture provides clear separation of concerns, with each tool handling a specific aspect of the analysis pipeline while maintaining a cohesive workflow.

2.1.3 Processing Strategy

The agent employs a balanced processing strategy that weighs performance against reliability. This strategy represents a key architectural decision made after evaluating different processing approaches for handling multiple violations within a single file.

Processing Strategy Options During the design phase, three main processing strategies were considered for handling multiple violations:

- **Fully Sequential Processing**: Each violation is processed one at a time, ensuring maximum reliability and predictability but potentially resulting in longer processing times for files with many violations.

- **Fully Parallel Processing**: All violations are processed concurrently, maximizing performance but introducing complexity in managing concurrent operations and potential reliability challenges under high load conditions.
- **Hybrid Processing Strategy**: A balanced approach that processes violations efficiently while maintaining system stability. This strategy provides significant performance improvements over sequential processing while ensuring reliable operation under various load conditions.

The hybrid processing strategy was selected as it provides the optimal balance between performance and reliability for production use. This strategy ensures that the agent can handle complex analysis tasks efficiently while maintaining system stability and providing consistent results.

2.1.4 Integration with LLM Infrastructure

The agent integrates with an internal AI platform that hosts multiple LLM models. The architecture is model-agnostic, enabling seamless adoption of newer models as they are released without requiring changes in the orchestration logic. LLM usage is isolated behind stable interfaces so higher-level logic remains unaffected. The analysis tools depend on the LLM for semantic understanding and code generation. This design ensures long-term maintainability and benefits from platform improvements without architectural change.

2.2 YouTube IDE Extension

The YouTube IDE Extension serves as the user-facing interface that seamlessly integrates the LLM Best Practices Agent into YouTube developers' daily workflow. Since YouTube developers are the primary target audience for this system, the YouTube IDE Extension was chosen as the natural entry point, leveraging their existing development environment and workflow patterns. This component is designed to provide intelligent, context-aware feedback while maintaining the responsiveness and familiarity that developers expect from their development environment. The feature becomes available when developers enable a user setting in the extension, and entry points only appear for files that belong to the internal YouTube framework for which we enforce best practices.

2.2.1 Extension Architecture

The YouTube IDE Extension serves as a lightweight client that orchestrates the interaction between developers and the AI analysis system. The architecture ensures responsiveness by delegating computationally intensive analysis to the specialized agent framework while handling user interface concerns, progress indication, and result presentation locally.

2.2.2 User-Triggered vs. Automatic Analysis Design Decision

A fundamental design decision for the IDE extension was whether to implement user-triggered analysis or automatic analysis. This choice significantly impacts user experience, system performance, and resource utilization.

The primary motivation for user-triggered analysis stems from the need to maintain developer productivity and system efficiency. LLM analysis is computationally expensive and resource-intensive, making continuous analysis impractical for maintaining IDE responsiveness. User-triggered analysis ensures that analysis occurs only when developers specifically request it, providing contextually relevant feedback at optimal moments without interrupting their workflow. This approach aligns with developer expectations of having control over their development environment while ensuring that computational resources are used efficiently.

Table III.1 summarizes the trade-offs:

Tableau III.1 – Comparison of User-Triggered vs. Automatic Analysis Approaches

Criteria	**User-Triggered**	**Automatic**
User Control	✓	×
Resource Efficiency	✓	×
IDE Performance	✓	×
Contextual Timing	✓	×
Discoverability	×	✓
Always Current	×	✓

Based on this analysis, the user-triggered approach was selected as it provides superior resource management, user control, and system performance. The trade-offs in discoverability and stale state management are addressed through intuitive UI design and comprehensive feedback mechanisms.

Stale State Challenge The user-triggered approach introduces a fundamental design challenge: maintaining the relevance and accuracy of analysis results as developers continue modifying their code. This challenge requires balancing system responsiveness with result accuracy, ensuring that feedback remains useful throughout the development process.

2.2.3 User Interface Design

The YouTube IDE Extension is designed around two core interaction patterns: entry points for initiating analysis and feedback mechanisms for presenting results.

Entry Points The YouTube IDE Extension provides multiple entry points to ensure accessibility and discoverability for different user preferences and workflows:

- **Visual Interface Integration**: Visual indicators are integrated into the development environment to provide immediate visibility of AI analysis availability, ensuring maximum discoverability while maintaining a clean interface.
- **Command-Based Access**: For developers who prefer keyboard-driven workflows, the extension provides command-based access through standard IDE navigation patterns, supporting both mouse-driven and keyboard-driven user interactions.

Feedback Mechanisms The YouTube IDE Extension employs three core feedback mechanisms designed to integrate seamlessly with existing development workflows:

- **Progress Indication**: Real-time status updates during analysis processing to maintain developer awareness and system transparency.
- **Violation Display**: Presentation of analysis results using familiar interface patterns that leverage developers' existing knowledge of standard feedback mechanisms.
- **Contextual Suggestions**: Interactive code solutions that appear when developers interact with violation markers, providing actionable recommendations.

2.2.4 User Interaction Flow

The user interaction flow with the YouTube IDE Extension follows a structured pattern from analysis initiation to optional fix application:

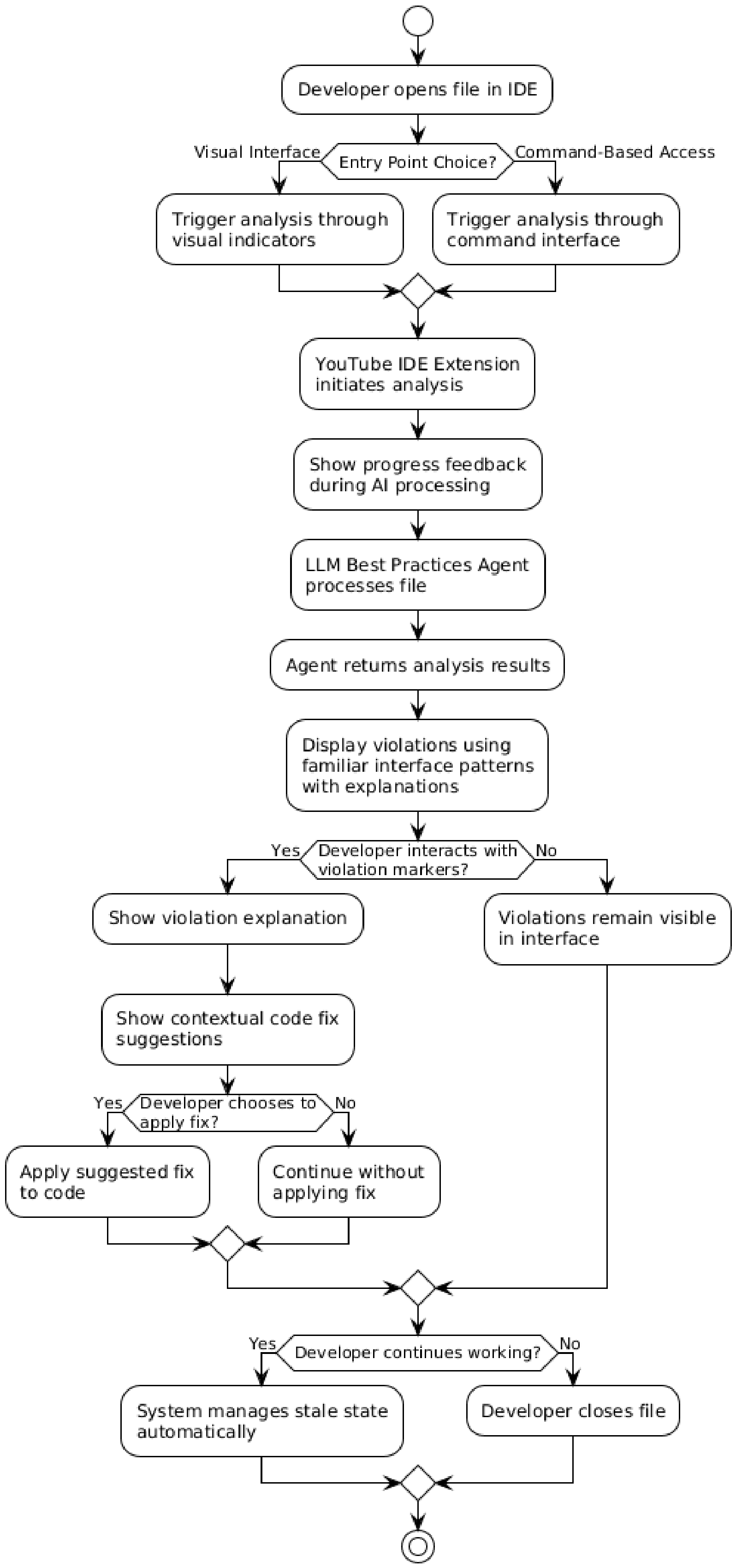

Developer opens file in IDE
Visual Interface
Entry Point Choice?
Command-Based Access
Trigger analysis through visual indicators
Trigger analysis through command interface
YouTube IDE Extension initiates analysis
Show progress feedback during AI processing
LLM Best Practices Agent processes file
Agent returns analysis results
Display violations using familiar interface patterns with explanations
Yes
Developer interacts with violation markers?
No
Show violation explanation
Violations remain visible in interface
Show contextual code fix suggestions
Yes
Developer chooses to apply fix?
No
Apply suggested fix to code
Continue without applying fix
Yes
Developer continues working?
No
System manages stale state automatically
Developer closes file

Figure III.5 – YouTube IDE Extension User Interaction Flow: Complete developer journey from analysis trigger to fix application

The flow demonstrates the core interaction pattern: developers initiate analysis through visual or command interfaces, receive progress feedback during processing, and interact with displayed violations to access explanations and optional fixes. This design ensures developers maintain control over their workflow while providing comprehensive feedback when requested.

3 Data Models and Interfaces

3.1 Input/Output Specification

The system's data models define the contracts between components, ensuring consistent communication and data exchange throughout the analysis pipeline. These interfaces establish clear boundaries between the IDE extension and the AI agent, enabling independent evolution of each component.

Analysis Request Format The YouTube IDE Extension sends analysis requests to the LLM Best Practices Agent using a minimal input format that identifies the target file for analysis. This design choice ensures that the agent can focus on its core responsibility of code analysis while maintaining security and proper workspace isolation. The request format includes the file path.

Analysis Response Format The agent returns a structured response containing the analysis results, error information, and optional metadata. The response format includes status information indicating success or failure, violation details with explanations and suggested fixes, and usage statistics for monitoring purposes. This standardized format ensures that the IDE extension can consistently process and display results regardless of the underlying analysis complexity.

3.2 Convention Data Management

The system's convention data model defines how YouTube framework best practices are structured, stored, and accessed throughout the analysis pipeline.

Convention Data Structure The convention data model captures best practice definitions as structured objects that support efficient programmatic access and analysis. Each convention definition includes essential metadata such as unique identifiers, descriptions, correct examples, and incorrect examples. This structure enables rapid lookup and context-specific retrieval during code analysis, with the design optimized for constant-time access patterns required by the agent's processing pipeline.

Storage Architecture Decision The system employs an in-memory storage approach using structured Python objects rather than external file-based or database storage. This design decision balances several architectural considerations:

- **Data Characteristics**: Conventions are static during runtime, requiring no dynamic updates, making in-memory storage appropriate for performance-critical analysis.
- **Performance Requirements**: In-memory access ensures ultra-low-latency retrieval for real-time analysis, eliminating I/O overhead during agent execution.
- **Simplicity & Reliability**: Structured Python objects provide type safety and eliminate parsing overhead while ensuring data integrity.
- **Resource Efficiency**: Conventions are loaded once at startup, minimizing runtime resource consumption and avoiding repeated file system access.
- **Architectural Flexibility**: The design allows future migration to external storage if multi-framework support or dynamic updates become necessary.

Integration Interfaces The system defines minimal, versioned boundaries that keep components decoupled:

- **IDE Extension Interface**: Contract between the YouTube IDE Extension and the LLM Best Practices Agent that defines the analysis request and response format, enabling independent evolution of UI and agent components.
- **Convention Access Interface**: Defines how the agent accesses convention definitions through in-memory lookup mechanisms, providing a stable boundary for data retrieval without external dependencies.
- **Monitoring Interface**: Captures usage statistics and performance metrics for system observability and evaluation purposes.

Conclusion

The architecture of the LLM Best Practices Agent is defined by three central design decisions. First, the adoption of the Executable Agent paradigm ensures deterministic execution, structured tool orchestration, and predictable performance, avoiding the drawbacks of open-ended reasoning loops. Second, the hybrid parallel–sequential processing strategy balances efficiency with interpretability, allowing analyses to scale while maintaining transparency in intermediate results. Finally, the IDE extension provides a seamless developer experience, integrating feedback, explanations, and fixes directly into familiar workflows while managing state and staleness automatically. Together, these pillars create a robust, cost-efficient, and developer-friendly system for embedding AI-driven best practice enforcement into the coding environment.

Chapter IV

Implementation

Summary

Introduction

This chapter presents the implementation of the LLM-powered best practices enforcement system, detailing how the architectural design from Chapter 3 was translated into a functional system. The implementation encompasses three primary components: the AI agent backend, the IDE extension frontend, and the communication infrastructure that connects them.

The chapter is structured to demonstrate the practical application of the theoretical framework established in previous chapters. It begins with the development environment and infrastructure, then presents the technology stack selections and their rationale, followed by detailed implementation of each system component.

1 Working Environment

1.1 Development Infrastructure

The implementation leverages Google's internal development infrastructure, providing support for large-scale software development. This infrastructure ensures security, scalability, and integration with existing YouTube development workflows.

1.1.1 Internal IDE

The development environment utilizes Google's internal IDE, which provides a development experience similar to Visual Studio Code but optimized for Google's internal infrastructure and security requirements.

1.1.2 Internal RPC Playground

The RPC Playground is Google's internal tool for testing and debugging Remote Procedure Call (RPC) services. This tool serves as a playground for sending RPC requests and was essential for developing and testing the communication protocol between the AI Agent and the YouTube IDE Extension.

1.1.3 Google Colab

Google Colab was used during early prototyping to iterate on prompt design, tool orchestration, and Executable Agent behaviors before production hardening. Colab provided hosted notebooks with on-demand compute (including GPUs/TPUs) and easy sharing for rapid experiments [8, 9]. It was part of the development environment rather than the deployed technology stack.

Figure IV.1 – Google Colab Logo

1.1.4 Internal Repository Integration

All code is stored and versioned within Google's internal repository system, enabling proper code review processes and collaboration.

1.2 Project Management and Documentation

1.2.1 Internal Version Control

The project utilizes Google's internal version control system, which provides Git-like functionality while ensuring compliance with internal security and access control requirements. Git is a fast, scalable, distributed version control system designed to handle everything from small to very large projects with speed and efficiency [10].

1.2.2 Internal Code Review Platform

Google's internal code review platform provides code review capabilities, ensuring code quality and knowledge sharing across development teams.

The code review platform includes:

- **Automated Review Suggestions**: AI-powered suggestions for code improvements, best practices, and potential issues.
- **Collaborative Review Process**: Tools for managing review workflows, assigning reviewers, and tracking review progress.
- **Integration with CI/CD**: Automatic triggering of builds and tests when code changes are submitted for review.

1.2.3 Internal Project Management System

The project management system provides project tracking, task management, and collaboration capabilities similar to Jira but optimized for Google's internal workflows. JIRA is a flexible issue tracking system that provides project management capabilities [11].

1.2.4 Google Docs

Google Docs was used for authoring and reviewing design documents, leveraging the internal built-in Approvals workflow to formalize stakeholder sign-off. The review process combined live comments, suggestions, and targeted approvals to ensure traceable decisions.

These project workflows ensured fast iteration, early detection of issues, and compliance with Google's security and code quality standards.

2 Technologies

This section presents the concrete technologies used to implement the system. We distinguish between industry-standard tools (e.g., Python, TypeScript, JSON) and internal platforms operated within Google (e.g., YouTube DevInfra Agent Framework, internal AI platform).

2.1 Backend Technologies (AI Agent)

2.1.1 Python Programming Language

Python serves as the primary programming language for the AI agent framework implementation. Python is a high-level, interpreted programming language known for its simplicity, readability, and library ecosystem [12]. The language's dynamic typing and support for artificial intelligence and machine learning libraries make it suitable for AI agent development [13].

Figure IV.2 – Python Programming Language Logo

Python's advantages for this implementation include:

- **AI/ML Ecosystem**: Extensive libraries for machine learning, natural language processing, and AI development.
- **Asynchronous Programming**: Built-in support for asynchronous programming patterns essential for handling concurrent requests.
- **JSON Processing**: Native support for JSON serialization and deserialization required for API communication.

2.1.2 YouTube DevInfra Agent Framework

The implementation utilizes the YouTube DevInfra Agent Framework — the serving infrastructure designed by YouTube for YouTube Developer Infrastructure agents. It underpins all YouTube agents, providing standardized execution, deployment, and operational primitives for LLM-powered applications.

This framework was selected because it natively supports the *Executable Agent* pattern and supplies built-in tool orchestration, workflow control, and production lifecycle management. Using it aligns the implementation with DevInfra standards and avoids rebuilding common agent infrastructure, allowing focus on best-practices enforcement logic.

2.1.3 Internal AI Platform

The system integrates with Google's internal AI platform, which hosts internal LLM models trained on Google's codebase, ensuring organization-specific knowledge and compliance with internal security requirements.

2.1.4 LLM Libraries and Frameworks

The internal agent framework utilizes several specialized libraries and frameworks for LLM interaction and agent development:

- **LLM Interaction Libraries**: Specialized libraries for communicating with internal LLM models, monitoring token usage, and optimizing API calls.
- **Agent Orchestration Libraries**: Libraries that provide the Executable Agent pattern implementation, tool registration, and workflow management.
- **Prompt Engineering Libraries**: Frameworks for constructing, optimizing, and managing prompts.

2.2 Frontend Technologies (IDE Extension)

2.2.1 TypeScript Programming Language

TypeScript is used for the frontend development of the YouTube IDE Extension. TypeScript is a strongly typed superset of JavaScript that compiles to plain JavaScript [14]. The language provides static type checking, which helps prevent runtime errors and improves code maintainability in large-scale applications.

TypeScript was chosen because we are building a feature inside an existing extension implemented in TypeScript, ensuring direct compatibility and reuse. Its static typing and interfaces improve maintainability and reduce runtime errors in complex UI state and service interactions. It also integrates seamlessly with VS Code API and other development environments.

Figure IV.3 – TypeScript Programming Language Logo

2.2.2 VS Code Extension API

The system integrates with Visual Studio Code through its Extension API. Visual Studio Code is a source-code editor developed by Microsoft, built on the Electron framework [15]. The VS Code Extension API provides capabilities for extending the editor's functionality.

Given that the internal IDE is VS Code–like, adopting the VS Code Extension API is the natural choice: it is natively supported within the environment (requiring no additional infrastructure), exposes the command, user-interface, and configuration interfaces required by the best-practices enforcement feature, and ensures compatibility with the existing extension ecosystem. In practice, the API provides the integration points necessary to implement the designed interaction flow without introducing custom runtime scaffolding.

2.2.3 JSON Data Format

JSON (JavaScript Object Notation) is used for data serialization and configuration management throughout the system. JSON is a lightweight, text-based data interchange format that is easy for humans to read and write [16].

JSON was chosen because the agent returns a string payload over RPC; encoding structured results as JSON preserves human readability and cross-language compatibility while satisfying the transport constraint.

Compared to gRPC with Protocol Buffers, JSON trades schema rigor and compact binary encoding for human-readability and ease of inspection. This project prioritizes operability within the IDE and debuggability during development, so JSON was preferred, with explicit schema validation applied at boundaries to mitigate the lack of strong compile-time contracts.

3 Realization

3.1 Agent Implementation

The AI Agent implementation follows the Executable Agent architecture pattern established in Chapter 3, providing deterministic execution with reliability and performance compared to alternative approaches. The implementation leverages Python's AI/ML ecosystem and Google's internal AI platform to create a scalable solution for YouTube framework best practices enforcement.

3.1.1 Agent Configuration and Initialization

The agent initialization process involves several critical configuration steps that ensure proper operation within the Google infrastructure:

- **Model Selection**: The agent is configured to use the latest internal Gemini-based model, specifically trained on Google's codebase.
- **Convention Loading**: At startup, the agent loads YouTube framework best practices from JSON configuration files stored in the internal repository.
- **Tool Registration**: All five specialized tools are registered with the agent framework, establishing the deterministic workflow pattern.
- **Error Handling Setup**: Comprehensive error handling mechanisms are initialized, including retry policies, timeout configurations, and fallback strategies.

Implementation Structure The agent is implemented as a class that exposes a single execute entry point responsible for orchestrating analysis. Tools are implemented as separate classes that expose a run method and encapsulate focused responsibilities (e.g., prompt construction, model invocation, response parsing). Public contracts specify inputs, outputs, and typed errors for each tool, while the agent coordinates them via a registry and passes dependency-injected helpers and shared utilities. Each tool also records token usage per invocation and emits lightweight metrics for observability. This class-based structure improves testability (mockable tool interfaces), maintainability (clear separation of concerns), and extensibility (tools can be replaced or added without modifying the orchestration logic).

3.1.2 Core Tools Implementation

The agent implements five specialized tools, each handling a specific aspect of the best practices analysis pipeline as defined in Chapter 3:

ReadFileFromWorkspace Tool Design contract: input = file path; output = full file content. The ReadFileFromWorkspace tool handles file system access and content retrieval operations. This tool implements error handling for common file system issues, including file not found errors, permission violations, and encoding problems. The tool handles multiple file formats and encodings that might arise in different development environments, providing error reporting to facilitate debugging and user feedback.

CodeAnalysisTool Design contract: input = file content; output = list of base violations. The CodeAnalysisTool serves as the core analysis engine, performing code analysis using LLM capabilities. The tool implements prompt engineering techniques to ensure consistent and accurate analysis results. The prompt construction process incorporates context-aware information, including file type, and relevant convention definitions.

Prompt Engineering Implementation The CodeAnalysisTool implements a multi-stage prompt construction process that ensures optimal LLM performance:

- **Context Injection**: The tool dynamically injects relevant YouTube framework conventions based on file type and detected patterns, ensuring that the LLM has access to the most relevant best practices.
- **Template System**: A template system allows for consistent prompt formatting while accommodating different file types and analysis contexts.

Performance Notes Selected optimizations implemented:

- **Caching Mechanism**: Convention definitions and prompt templates are cached in memory to reduce repeated processing overhead.
- **Response Parsing**: JSON parsing with error recovery ensures handling of LLM responses.

ViolationExplanationTool Design contract: input = base violation; output = natural-language explanation. The ViolationExplanationTool generates human-readable explanations for identified violations. This tool implements natural language generation techniques to provide clear, educational explanations that help developers understand not only what violations exist, but why they are problematic and how they impact code quality.

The explanation generation process incorporates contextual information about the violation, surrounding code, and relevant best practices. This contextual approach ensures that explanations are specific, relevant, and actionable for developers.

CodeFixTool Design contract: input = base violation + explanation; output = suggested fix. The CodeFixTool provides actionable fix suggestions for identified violations. The tool implements code generation techniques that produce fixes designed to be:

- **Safe**: Only modify internal implementation without breaking public APIs
- **Self-contained**: Require no additional changes in other files
- **Contextual**: Take into account the specific code context and framework patterns
- **Educational**: Include comments explaining the reasoning behind the fix

The generated fixes include explanatory comments to help developers understand the reasoning behind the suggested changes.

The fix generation process prioritizes safety and correctness, ensuring that suggested changes maintain the original functionality while addressing the identified violations. The tool also considers the broader codebase context to ensure that fixes align with existing patterns and conventions.

Finish Tool Design contract: input = violation results + explanations + fixes; output = structured response format. The Finish tool serves as the consolidation component, aggregating all analysis results into a structured output format. This tool implements result validation, ensuring that the output contains all necessary information for the IDE extension to display results effectively.

The consolidation process includes deduplication of overlapping violations, merging of related issues, and formatting of results for optimal presentation.

3.1.3 Processing Strategy Architecture

The agent implements the hybrid parallel processing strategy with concurrency limiting as described in Chapter 3. This approach balances the benefits of concurrent processing with the need for system stability and resource management. The implementation uses **semaphore-based concurrency control** to limit the number of concurrent LLM calls, preventing resource saturation while maintaining optimal performance.

The internal workflow is illustrated in the activity diagram below:

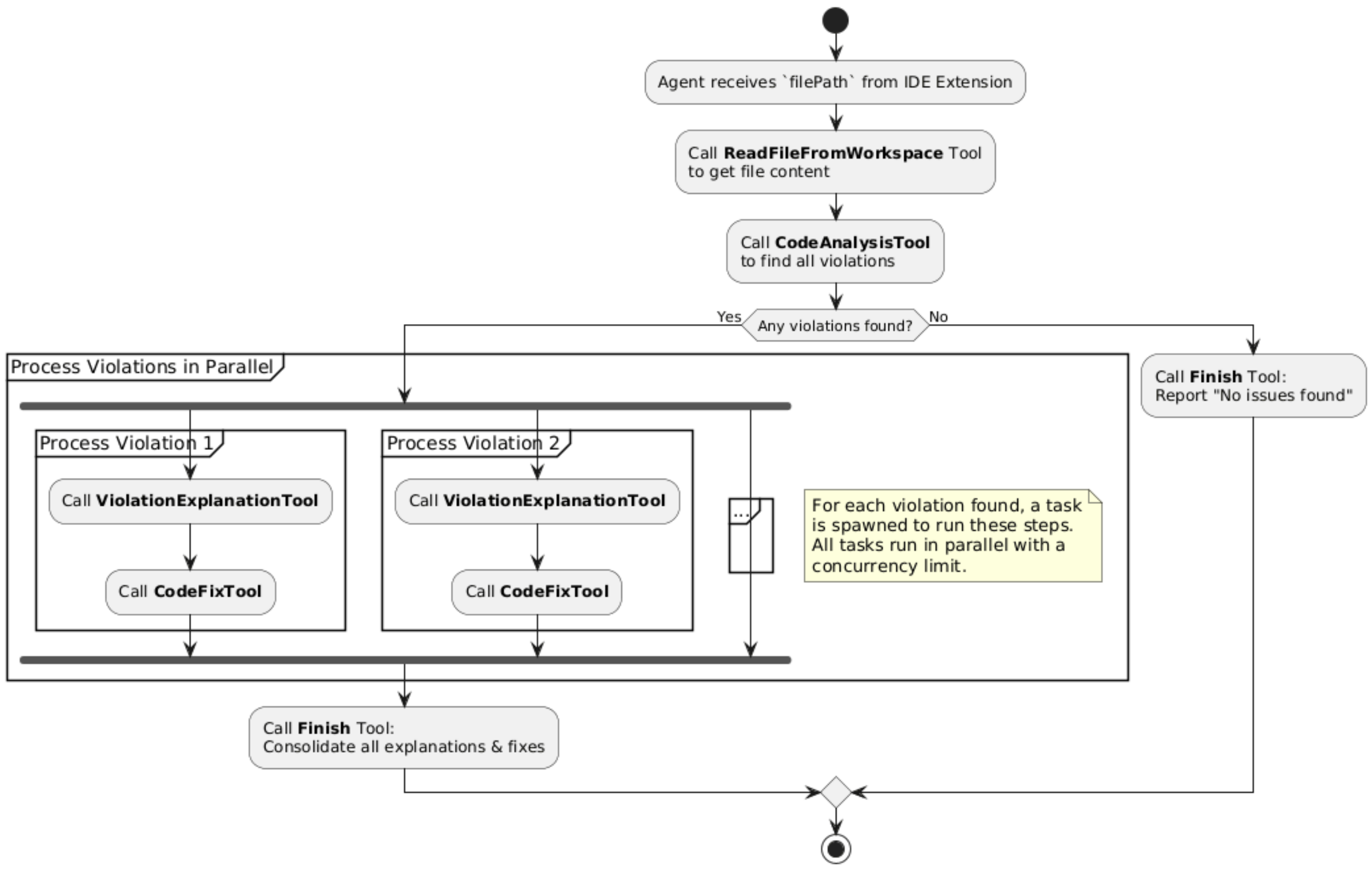

Agent receives `filePath` from IDE Extension
Call ReadFileFromWorkspace Tool to get file content
Call CodeAnalysisTool to find all violations
Yes
Any violations found?
No
Process Violations in Parallel
Process Violation 1
Call ViolationExplanationTool
Call CodeFixTool
Process Violation 2
Call ViolationExplanationTool
Call CodeFixTool
...
For each violation found, a task is spawned to run these steps. All tasks run in parallel with a concurrency limit.
Call Finish Tool: Report "No issues found"
Call Finish Tool: Consolidate all explanations & fixes

Figure IV.4 – Agent Processing Activity Diagram

This activity diagram illustrates the hybrid processing strategy in action, showing how the agent balances sequential and parallel processing to optimize performance and reliability.

This workflow represents the processing scenario that occurs when the agent analyzes a file with multiple violations.

Sequential Initial Processing The agent begins with sequential steps to ensure that the complete code context is available before any analysis begins, and all violations are identified before parallel processing starts.

Parallel Violation Processing When violations are found, the agent employs the hybrid processing strategy. Each violation is processed independently, with explanation generation and fix creation happening concurrently across multiple violations while maintaining system stability.

Result Consolidation The workflow concludes with result consolidation, ensuring that all processing results are properly aggregated into a coherent response format for the YouTube IDE Extension.

Concurrency Control Implementation The implementation uses a semaphore-based concurrency control system with concurrency limiting to limit the number of concurrent LLM calls. A default concurrency limit is applied per worker pool, and excess tasks are queued in a FIFO (First In, First Out) queue to prevent resource saturation. This ensures that the system remains stable even under high load conditions.

3.1.4 Error Handling and Resilience

The system implements error isolation mechanisms where failures in one violation processing task do not stop the processing of other violations. Each violation is processed independently, and partial results are persisted even when individual tasks fail. This approach ensures that developers receive feedback even when some analysis steps encounter errors.

The implementation includes retry mechanisms with exponential backoff for transient failures, ensuring operation in production environments. The error handling system implements specific error types for different failure scenarios, including file system errors, network communication errors, LLM processing errors, and agent orchestration errors. Each error type includes detailed error information and suggested recovery actions to facilitate debugging and user support.

3.1.5 Convention Data Management

The convention data management system implements loading, caching, and retrieval mechanisms for YouTube framework best practices. The conventions are stored as Python objects in an array, each containing a unique identifier, description, correct example, and incorrect example. For efficient runtime access, the system constructs an in-memory map keyed by convention ID, allowing the agent tools to retrieve only the relevant convention on demand.

Loading and Initialization At startup, all convention objects are loaded into memory from the Python array. A dictionary (map) is created with convention IDs as keys and convention objects as values, providing constant-time access for subsequent tool invocations.

Memory Caching and Access This in-memory caching strategy ensures low-latency access during code analysis:

- **Efficient Lookup**: Tools retrieve conventions by ID from the map, avoiding iteration over the full array.
- **Dynamic Selection**: Only conventions relevant to the current file type and analysis context are queried, minimizing unnecessary data processing.
- **Lightweight and Fast**: The cache resides entirely in memory, requiring no external services, and supports rapid retrieval during concurrent tool executions.

3.2 Extension Integration

3.2.1 Extension Architecture

The IDE Extension implements a layered architecture that integrates with the overall system through three distinct layers: the Extension layer containing user-facing components, a Proxy layer for authentication and request routing, and the Backend layer hosting AI agent services.

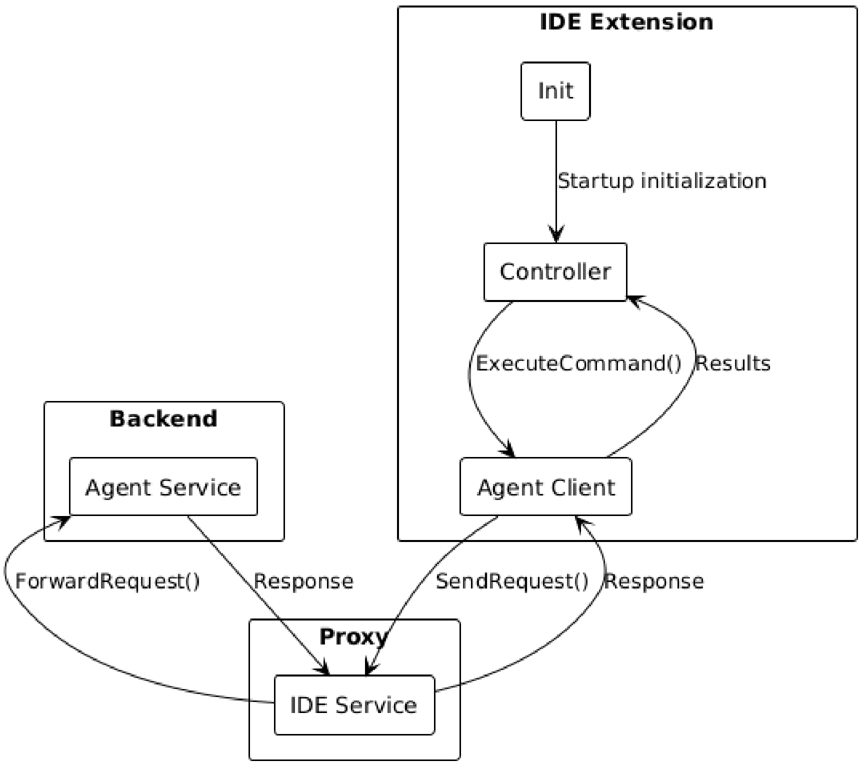


Figure IV.5 – System Architecture: IDE Extension, Proxy, and Backend Communication Flow

The architecture follows a clear request-response pattern where user interactions trigger analysis requests that flow through the IDE Service proxy for authentication and authorization, then to the Agent Service in the backend for processing. Responses follow the same path in reverse, ensuring secure and authenticated communication throughout the entire pipeline while maintaining clear separation of responsibilities between layers.

3.2.2 Extension Components

The extension's internal architecture consists of three core components that work together to provide seamless integration with the development environment, as illustrated in Figure IV.5.

Init Component: Handles extension initialization, reading user settings, registering commands and editor actions, and performing health checks. The component ensures proper setup of all dependencies.

Controller: Centralizes all UI-related state and orchestrates interactions between components. The Controller processes commands, manages notifications, routes analysis results, renders diagnostics and hover-based suggestions, and coordinates stale-state transitions to ensure consistent behavior across all entry points.

AgentClient: Manages communication with the backend services through the proxy layer. The component handles request formatting, implements retry logic with exponential backoff, manages timeouts, and processes responses from the AI agent.

3.2.3 User Interaction

The feature is controlled through a dedicated **user setting**. This setting appears as a simple checkbox: when enabled, the feature becomes available in the IDE; when disabled, it is entirely hidden from the interface. This ensures that developers can opt in seamlessly without cluttering the environment for those who do not use the feature.

The primary entry point for triggering analysis is an **Editor Action** integrated into the file title bar (Figure IV.6). This placement ensures high visibility and aligns naturally with the developer's workflow when working on individual files.

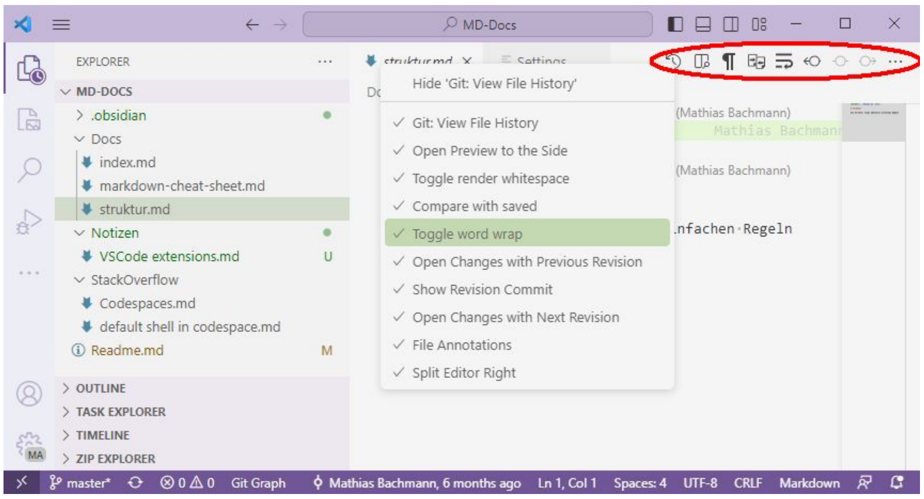


Figure IV.6 – VS Code Interface: Editor Actions (Illustrative).

An additional entry point is provided through the **Command Palette**, which can be invoked using `Ctrl+Shift+P` (or `Cmd+Shift+P` on macOS). This pathway makes the feature equally accessible to developers who prefer keyboard-driven workflows and ensures discoverability for new users exploring available commands (Figure IV.7).

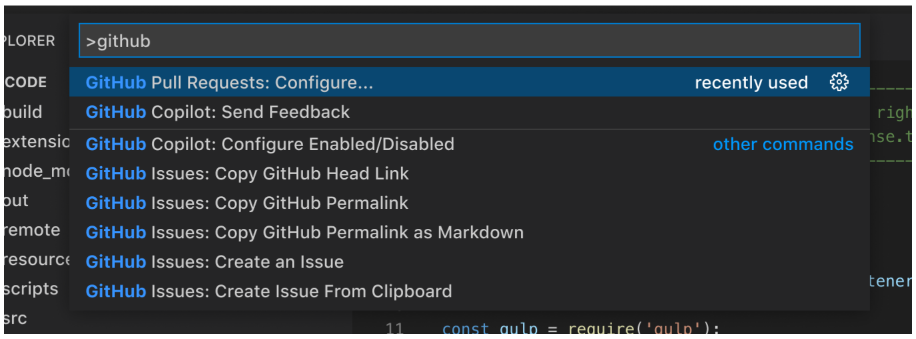
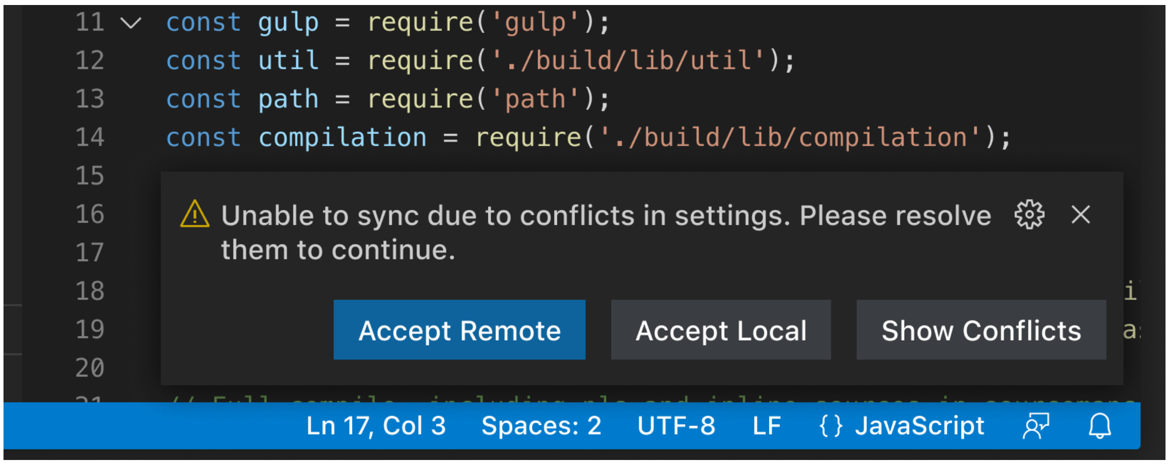

>github
GitHub Pull Requests: Configure...
recently used
GitHub Copilot: Send Feedback
GitHub Copilot: Configure Enabled/Disabled
other commands
GitHub Issues: Copy GitHub Head Link
GitHub Issues: Copy GitHub Permalink
GitHub Issues: Copy GitHub Permalink as Markdown
GitHub Issues: Create an Issue
GitHub Issues: Create Issue From Clipboard

Figure IV.7 – VS Code Command Palette (Illustrative).

Once analysis is triggered, the extension provides immediate feedback via **VS Code notifications** (Figure IV.8). These notifications confirm that a request has been received, update progress, and display clear error messages if issues occur. This ensures transparent communication throughout the request lifecycle.

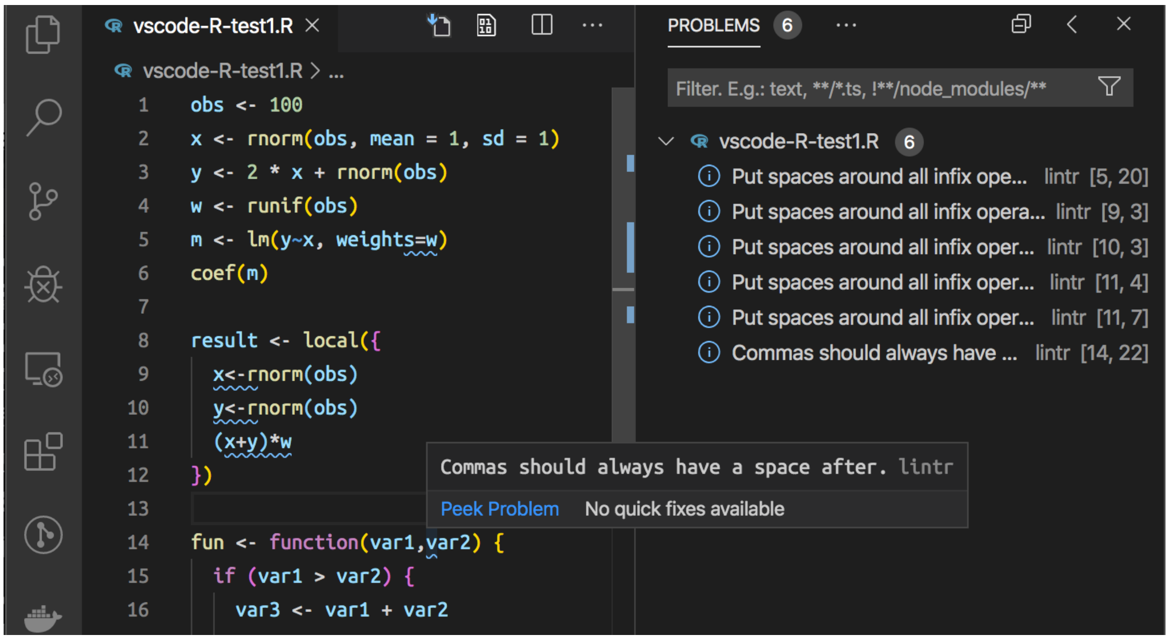


```
const gulp = require('gulp');
const util = require('./build/lib/util');
const path = require('path');
const compilation = require('./build/lib/compilation');
```

⚠ Unable to sync due to conflicts in settings. Please resolve them to continue.

Figure IV.8 – VS Code Notification Interface (Illustrative).

The results of the analysis are surfaced through VS Code's native **diagnostic system**. Violations appear in the **Problems panel** and are underlined directly in the editor, marking the exact range of code that violates a best practice (Figure IV.9). Hovering over the highlighted code reveals the diagnostic explanation, helping developers quickly understand the issue in context.



```
obs <- 100
x <- rnorm(obs, mean = 1, sd = 1)
y <- 2 * x + rnorm(obs)
w <- runif(obs)
m <- lm(y~x, weights=w)
coef(m)

result <- local({
  x<-rnorm(obs)
  y<-rnorm(obs)
  (x+y)*w
})

fun <- function(var1,var2) {
  if (var1 > var2) {
    var3 <- var1 + var2
```

```
ⓘ Put spaces around all infix ope...   lintr [5, 20]
ⓘ Put spaces around all infix opera... lintr [9, 3]
ⓘ Put spaces around all infix oper...  lintr [10, 3]
ⓘ Put spaces around all infix oper...  lintr [11, 4]
ⓘ Put spaces around all infix oper...  lintr [11, 7]
ⓘ Commas should always have ...        lintr [14, 22]
```

```
Commas should always have a space after. lintr
```

Figure IV.9 – VS Code Diagnostics Interface (Illustrative).

For more detailed guidance, an integrated **hover provider** presents formatted fix suggestions directly within the editor. This approach allows fixes to be displayed with proper syntax highlighting and inline code snippets, offering a clear and actionable path to resolution without leaving the development workflow.

3.2.4 Stale Diagnostics Handling

One of the most challenging aspects of IDE integration is maintaining diagnostic accuracy as developers continuously modify their code. The extension implements a sophisticated two-tiered system that balances immediate responsiveness with accurate analysis results.

The Challenge Traditional diagnostic systems struggle with code that changes rapidly, often displaying outdated information that confuses developers and reduces trust in the tool. The challenge is to provide immediate visual feedback while ensuring that diagnostics remain accurate and relevant to the current code state.

Two-Tiered Solution The extension handles stale diagnostics using a two-tiered approach that separates immediate responsiveness from precise re-anchoring:

Tier 1 - Instant Adjustment: Provides immediate feedback on every keystroke. Diagnostics are shifted based on simple text edits and marked as [Outdated] if the flagged code itself is edited. This ensures high **responsiveness** without impacting performance.

Tier 2 - Debounced Re-anchoring: Activates after a 1-second pause in typing to improve diagnostic **accuracy**. The process involves:

1. **Fingerprint**: Creates a contextual hash from the code and its surrounding context to identify exact matches.
2. **Scan with Regex**: Finds all possible text matches across the document.
3. **Re-anchor**: Moves the diagnostic to the correct new location based on the highest match score.

This two-tiered strategy balances responsiveness with accuracy, ensuring developers receive timely feedback while maintaining the integrity of diagnostics even during active code editing.

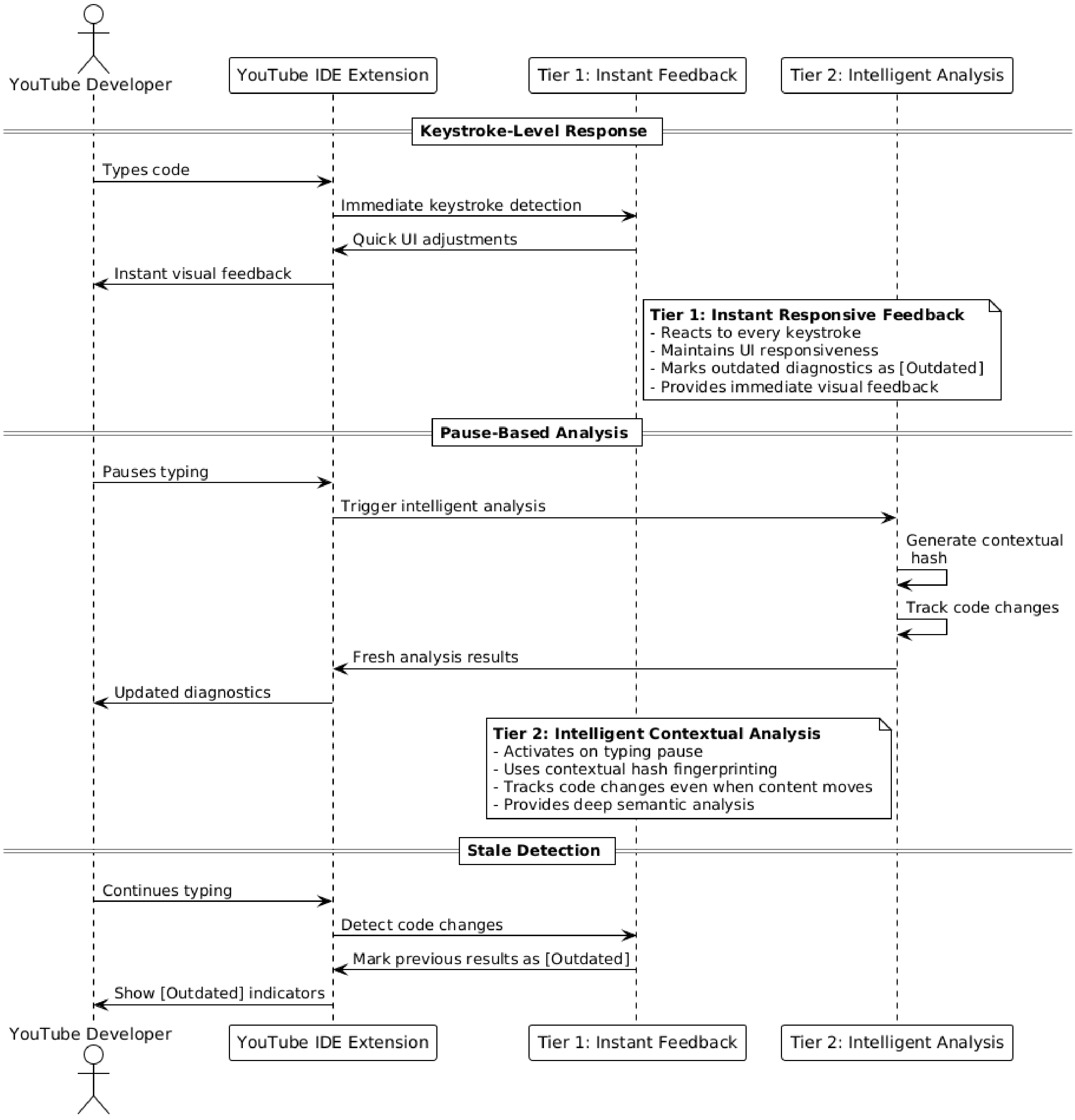

YouTube Developer
YouTube IDE Extension
Tier 1: Instant Feedback
Tier 2: Intelligent Analysis
Keystroke-Level Response
Types code
Immediate keystroke detection
Quick UI adjustments
Instant visual feedback
Tier 1: Instant Responsive Feedback
- Reacts to every keystroke
- Maintains UI responsiveness
- Marks outdated diagnostics as [Outdated]
- Provides immediate visual feedback
Pause-Based Analysis
Pauses typing
Trigger intelligent analysis
Generate contextual hash
Track code changes
Fresh analysis results
Updated diagnostics
Tier 2: Intelligent Contextual Analysis
- Activates on typing pause
- Uses contextual hash fingerprinting
- Tracks code changes even when content moves
- Provides deep semantic analysis
Stale Detection
Continues typing
Detect code changes
Mark previous results as [Outdated]
Show [Outdated] indicators

Figure IV.10 – Stale Diagnostics Handling: Two-Tiered System (Illustrative)

This approach ensures that developers receive immediate visual feedback while maintaining diagnostic accuracy through intelligent analysis timing and state management, representing a significant technical contribution to IDE integration challenges.

3.2.5 Resilience and Communication

The extension implements comprehensive error handling and resilience mechanisms to ensure reliable operation in production environments. The communication system uses internal RPC infrastructure with JSON payloads for debugging and cross-language compatibility.

Error handling follows fault-tolerant design principles with multi-level error isolation, ensuring that failures in one component do not cascade to others. The system implements specific error types for different failure scenarios, including network communication errors, service unavailability, and timeout errors, with detailed error information and suggested recovery actions.

Retry mechanisms with exponential backoff and bounded attempts ensure operation under transient failure conditions, while timeout management prevents indefinite waiting periods and maintains responsive user experience. The implementation includes request validation and response parsing to prevent communication errors and ensure data integrity throughout the analysis pipeline.

Conclusion

This chapter presented the implementation of the LLM-powered best practices enforcement system, detailing the development environment, technology stack, and practical realization. The implementation successfully translates the architectural design from Chapter 3 into a functional system that integrates AI agent capabilities with IDE extension functionality.

The implementation establishes a solid foundation for real-world deployment and future enhancements, setting the stage for the performance evaluation presented in Chapter 5.

Chapter V

Evaluation

Summary

Introduction

This chapter presents a comprehensive evaluation of the LLM-powered best practices enforcement system that drove the architectural decisions described in Chapter 3 and the implementation choices detailed in Chapter 4. The evaluation focuses on comparing three distinct agent architectures to determine the optimal approach for production deployment, analyzing their performance across accuracy, latency, and cost metrics.

The results presented in this chapter directly informed the design decisions and implementation strategies outlined in previous chapters, demonstrating how data-driven evaluation shaped the final system architecture.

1 Evaluation Methodology

1.1 Test Suite Framework

We created a comprehensive suite of 12 test cases covering a wide range of scenarios to evaluate the three agent architectures. This test suite provides diverse complexity levels and violation patterns to ensure robust evaluation across different use cases.

1.2 LLM-as-a-Judge Evaluation Approach

The most novel aspect of our evaluation framework is the ”LLM-as-a-Judge” methodology. A significant challenge in testing LLMs is the inherent variability in their outputs, making simple text comparison methods too brittle for reliable assessment.

Our solution employs another LLM as an impartial judge that semantically compares the agent’s response to the ideal answer. This approach understands the meaning behind the words rather than requiring exact text matches, providing a much more realistic and robust measure of quality.

1.3 Key Metrics

With this evaluation framework in place, we track three critical metrics that determine system viability:

- **Accuracy**: Determined by our LLM judge through semantic comparison
- **Latency**: Processing speed measured in seconds
- **Cost**: Token consumption and associated computational costs

2 Agent Architecture Comparison

2.1 Architecture Variants

Three distinct agent architectures were implemented and evaluated to inform the design decisions presented in Chapter 3. These architectures represent different approaches to balancing performance, reliability, and resource efficiency:

- **Sequential Executable**: Processes violations one by one, ensuring deterministic execution and maximum reliability. This approach was evaluated to establish a baseline for reliability and consistency.

- **Parallel Executable**: Processes all violations concurrently using asyncio.gather, maximizing throughput through parallelization. This architecture was tested to assess the potential performance gains and associated risks.
- **ReAct Agent**: Uses a ReAct loop with multi-step reasoning, representing the traditional agent paradigm. This approach was included to evaluate the effectiveness of dynamic reasoning versus predefined execution patterns.

The evaluation of these three architectures directly informed the hybrid approach described in Chapter 3, where we ultimately selected the parallel executable architecture with concurrency limiting based on the performance and reliability data presented in this chapter.

3 Performance Analysis Results

3.1 Single-Violation File Analysis

For a simple file with only one violation (TC8_PROPS_MISSING_EXPORT), the performance differences are stark:

Tableau V.1 – Single-Violation File Performance Comparison

Architecture	**Latency**	**Input Tokens**	**Output Tokens**
Parallel Executable	4.1s	5,157	331
Sequential Executable	6.1s	5,157	331
ReAct Agent	15.3s	37,989	1,523

Key Takeaways:

- **Parallel vs. Sequential**: For a simple case, the overhead of asyncio.gather is negligible, making the parallel agent about 33% faster than the sequential one. Token usage is identical.
- **Executable vs. ReAct**: The executable agents are vastly more efficient. The ReAct agent is 2.7x slower than the sequential agent and uses 7.4x more input tokens to solve the same simple problem.

3.2 Full Evaluation Suite Analysis (12 Cases)

When scaling up to the full test suite, the reliability of the parallel agent becomes the central issue. The following charts illustrate the comprehensive performance comparison across all 12 test cases:

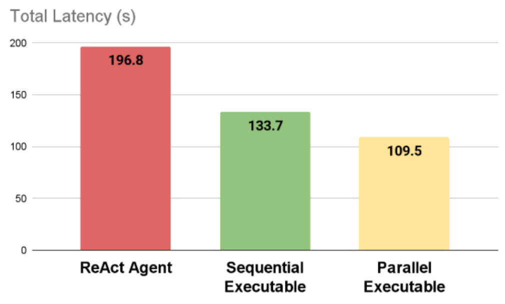


Figure V.1 – Latency Performance Comparison Across 12 Test Cases

Latency Performance (Chart 1): The latency chart clearly demonstrates the parallel agent's speed advantage, completing the entire test suite about 22% faster than the sequential agent. However, this performance comes with a critical caveat - the parallel agent's success is conditional and depends on the total number of concurrent API calls.

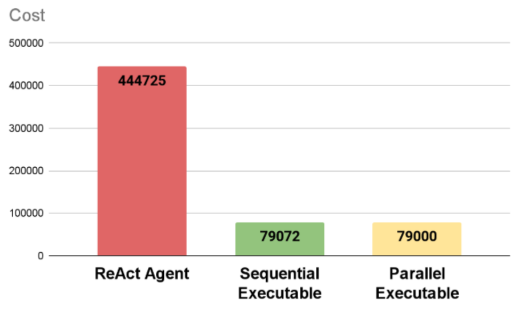


Figure V.2 – Cost Analysis Across 12 Test Cases

Cost Analysis (Chart 2): The cost comparison reveals the ReAct agent's fundamental inefficiency, consuming nearly six times more tokens than the executable agents. This massive cost differential makes the ReAct approach non-viable for production deployment, effectively eliminating it from consideration.

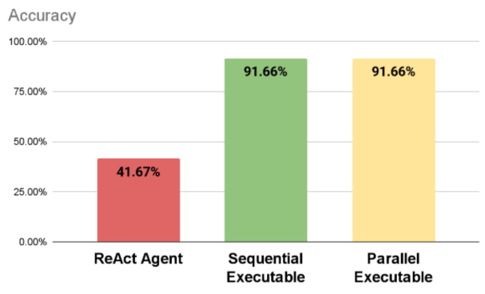


Figure V.3 – Accuracy Performance Across 12 Test Cases

Accuracy Performance (Chart 3): While both executable agents achieve identical accuracy rates of 91.66%, the parallel agent's reliability becomes the deciding factor. The sequential agent maintains 100% reliability across all test runs, while the parallel agent suffers from OverLimitException and DEADLINE_EXCEEDED errors under heavy load.

Key Takeaways:

- **The Reliability Flaw**: While the parallel agent was 22% faster in its successful run, it has also been observed to crash with OverLimitException when encountering files with many violations. Its success is conditional and depends on the total number of concurrent API calls, which can easily exceed service quotas.
- **The Stability of Sequential**: The sequential agent is the only architecture that has proven to be 100% reliable across all test runs.
- **ReAct's Inefficiency**: The ReAct agent remains non-viable due to its low accuracy and extremely high cost.

3.3 Detailed Performance Analysis

Let's examine the data we found when comparing our three candidates:

Latency Analysis: If this were just about being fast, the parallel agent would be the clear winner. It had the lowest latency, completing the entire test suite about 22% faster than the sequential one. But as we'll see, the lowest latency isn't the whole story.

Cost Analysis: This is where the ReAct agent fell apart. Because of its ReAct loop—where it has to 'think' step-by-step about what to do next—it consumed a massive number of tokens. It cost nearly six times more to do the exact same job. That incredible inefficiency made it non-viable, so right away, we knew ReAct was out. This narrowed our choice down to the two executable models.

Accuracy Analysis: Looking at the results, the Sequential agent was very accurate, with a solid 91.7% success rate. And the Parallel agent appears to match it perfectly. But there's a critical catch. That high accuracy is only for the runs where it didn't crash. We discovered that under heavy load, its fully parallel nature would cause it to fail completely. And the agent with the lowest latency is useless if it's not reliable.

So, the data left us with a clear puzzle: ReAct was too expensive. Sequential was reliable but had higher latency. And Parallel had the lowest latency but was critically unstable. The question became: how could we get the speed we wanted, without all the risk?

4 Final Recommendation

4.1 Implement the Parallel Agent with Concurrency Limiting

The core decision comes down to a trade-off between the parallel agent's raw speed and its reliability under load. Our findings show that while the fully parallel agent is the fastest in simple cases, it is prone to critical failures when processing files with many violations.

This evaluation directly drove the implementation strategy described in Chapter 4, where we implemented the parallel executable architecture with semaphore-based concurrency control to address the reliability issues identified in our testing.

The most recent test runs surfaced a DEADLINE_EXCEEDED error. This occurs because flooding the backend with too many simultaneous requests overwhelms the service, preventing it from responding before the client-side timeout is reached. This is a more insidious failure than a simple quota error (OverLimitException) because it points to systemic overload.

For a production tool, predictability and stability are paramount. A user encountering a hard error is a significantly worse experience than waiting a few extra seconds for a guaranteed result.

Therefore, the recommendation is to keep the parallel architecture but control its concurrency. By using a semaphore to limit the number of in-flight requests, we can retain most of the performance gains of parallelism while ensuring the agent remains stable and reliable, even under heavy load.

This approach gives us the best of both worlds: the speed of parallelism and the reliability of a sequential process.

Conclusion

This chapter presented a comprehensive evaluation of three agent architectures for the LLM-powered best practices enforcement system. Through our novel "LLM-as-a-Judge" evaluation methodology and comprehensive 12-test-case suite, we demonstrated that the parallel executable agent with concurrency limiting provides the optimal balance between performance and reliability.

The evaluation revealed that while the ReAct agent was eliminated due to its excessive cost and low accuracy, the choice between sequential and parallel executable agents required careful consideration of the reliability-speed trade-off. Our final recommendation of implementing the parallel agent with concurrency limiting achieves the best of both worlds: the speed of parallelism and the reliability of sequential processing.

This data-driven evaluation process directly informed the architectural decisions presented in Chapter 3 and guided the implementation strategies detailed in Chapter 4. The systematic comparison of agent architectures, performance analysis, and reliability assessment provided the empirical foundation for selecting the optimal approach, ensuring that design choices were based on concrete evidence rather than assumptions. This evaluation methodology demonstrates how rigorous testing and analysis can drive effective system design and implementation decisions.

Conclusion and Perspectives

This graduation project has successfully developed and implemented an intelligent assistance system integrated into YouTube's internal development environment, aimed at enforcing internal framework-specific best practices in real-time. The system addresses a major challenge in large-scale development environments: the lack of real-time feedback on internal framework-specific best practices. While existing tools focus on general syntax or public frameworks, our solution provides contextual and intelligent assistance directly within the developer's workflow, reducing technical debt and improving code consistency.

The main contribution consists of integrating an AI agent based on Large Language Models (LLMs) into the YouTube development environment. The system comprises an executable AI agent using the "Executable Agent" architecture that orchestrates five specialized tools for code analysis, violation explanation, and fix generation, a YouTube IDE extension that provides an intuitive user interface with multiple entry points, and a sophisticated two-tier state management system for handling stale diagnostics, balancing immediate responsiveness with analysis precision.

Development was carried out using a modern technology stack including Python for the AI agent implementation, the YouTube DevInfra Agent Framework for infrastructure, Google's internal AI platform for LLM models, TypeScript for the IDE extension, and VS Code Extension API for integration. The project followed an agile approach with a Kanban workflow, structured in clear phases from onboarding through implementation, testing, and optimization.

Several significant technical challenges were overcome during the project. Stale state management required a sophisticated two-tier system with instant adjustment for responsiveness and debounced re-anchoring for precision. Performance optimization involved evaluating three different agent architectures, resulting in the choice of a parallel agent with concurrency limiting for optimal performance and stability. AI quality evaluation was complex due to LLM output variability, leading to the development of the "LLM-as-a-Judge" solution for robust semantic evaluation.

The primary future focus is implementing a Tiered Analysis Approach to make the tool faster and more efficient. Currently, our agent uses its powerful LLM brain for everything, which is excellent for complex problems but overkill for simple issues. The vision is to add a super-fast linter that acts as a first pass, instantly catching easy, clear-cut issues, while only complex, subjective problems would be passed to the LLM agent. This approach provides the best of both worlds: instant speed for easy wins and deep intelligence for hard cases, resulting in a faster, cheaper, and better developer experience. The technical groundwork is already in place, as a temmate is currently building the super-fast, rule-based linter, and we have established a clear integration plan.

This project has demonstrated the feasibility and effectiveness of integrating AI agents into development environments for enforcing internal framework-specific best practices. The solution addresses a real need in the YouTube development ecosystem and provides a solid foundation for future developments, positioning this work as an important contribution to improving development tools and adopting AI in software development processes.

References

[1] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Brian Marick, et al. *Manifesto for agile software development.* Agile Alliance (2001). 7

[2] Robert C Martin. *Agile software development: principles, patterns, and practices.* Prentice Hall PTR (2003). 7

[3] David J Anderson. *Kanban: Successful evolutionary change for your technology business.* Blue Hole Press (2010). 7

[4] Henrik Kniberg and Mattias Skarin. *Kanban and Scrum-making the most of both.* Lulu. com (2011). 7

[5] Google. Google's ai code generation statistics. Google I/O 2024 Keynote, (2024). Accessed: 2024-12-19. 16

[6] Google. Google developer survey: Ai in software development. Google Developer Blog, (2024). Accessed: 2024-12-19. 16

[7] Microsoft Azure Architecture Center. Ai agent design patterns. `https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/ai-agent-design-patterns`, (2024). Accessed: 2025-09-07. 33

[8] Google Research. Google colab. `https://colab.research.google.com/`, (2017). Accessed: 2024-12-19. 44

[9] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, et al. *Jupyter notebooks—a publishing format for reproducible computational workflows.* Positioning and Power in Academic Publishing: Players, Agents and Agendas pages 87–90 (2014). 44

[10] Linus Torvalds and Junio Hamano. *Git: A fast, scalable, distributed version control system.* Proceedings of the Linux Symposium **1**, 3–10 (2005). 45

[11] Atlassian. *Jira: A flexible issue tracking system.* Atlassian Documentation (2002). 45

[12] Guido Van Rossum and Fred L Drake Jr. *Python reference manual.* Amsterdam: CWI (Centre for Mathematics and Computer Science) (1995). 46

[13] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. *Scikit-learn: Machine learning in python*. Journal of machine learning research **12**(Oct), 2825–2830 (2011). 46

[14] Gavin Bierman, Martin Abadi, and Mads Torgersen. Understanding typescript. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 257–268 (2014). 47

[15] Fernando Castor, Eduardo Figueiredo, Nélio Cacho, Breno Sena, Leonardo Teixeira, Gustavo Pinto, and Breno Fonseca. Visual studio code: A new way of developing web applications. In *Proceedings of the 2016 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 1–2 (2016). 48

[16] Douglas Crockford. The application/json media type for javascript object notation (json), (2006). 48

Appendix : Miscellaneous remarks