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 : **30/09/2025**

JURY

Ms. President Ghada GASMI (President)
Ms. Reviewer Asma BEN HASSOUNA (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

Google

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.

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 : Génie Logiciel, Intelligence Artificielle Générative, 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: Software Engineering, Generative AI, IDE Integration, Code Quality, Developer Productivity

Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
CI	Continuous Integration
CI/CD	Continuous Integration/Continuous Deployment
DL	Deep Learning
FIFO	First In, First Out
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LLM	Large Language Model
ML	Machine Learning
QA	Quality Assurance
RPC	Remote Procedure Call
SDLC	Software Development Life Cycle
VS	Visual Studio

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 situating the project within its organizational context and outlining the adopted methodology, then builds the conceptual background and reviews related work to motivate and refine the requirements. It next articulates the system's design and architecture, before discussing the evaluation that informed the final approach and detailing the resulting implementation of the AI agent and its IDE integration.

Chapter I

Project Overview

Introduction

This opening chapter establishes the foundation of the 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

1.1.1 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 establishes 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 1.1 shows the Google logo.

Figure 1.1 – Google Logo

1.1.2 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. Figure 1.2 provides an overview of some Google products.

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

Figure 1.2 – Overview of some Google products

1.1.3 Focus Area

1.1.3.1 YouTube

Acquired by Google in 2006 [**?**], YouTube becomes 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 1.3 displays the YouTube logo.

Figure 1.3 – YouTube Logo

1.1.3.2 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

1.2.1 Project Context

This project develops 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 explores 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 end-of-studies-internship project required for obtaining the software engineering degree at the National Institute of Applied Science and Technology, providing both academic and practical significance.

1.2.2 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.

1.2.3 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

1.3.1 Agile Development Approach

Agile software development, as defined by Beck et al. [**?**], 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 is adopted to support iterative development and maintain flexibility in responding to evolving requirements. According to Martin [**?**], 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 are incorporated throughout the process to maintain high quality, while frequent collaboration provides clarity and shared ownership of outcomes. Agile principles complement the focus on engineering excellence, including rigorous design reviews, thorough testing, robust code reviews, and DevOps-enabled automation.

1.3.2 Kanban Workflow

Kanban, as described by Anderson [**?**], 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, is managed using this Kanban workflow. According to Kniberg and Skarin [**?**], Kanban enables teams to visualize tasks and limit work in progress (WIP), preventing bottlenecks and allowing quick focus shifts to urgent issues when necessary. Work progresses in a pull-based manner, with explicit WIP limits and clear policies per column, and is structured into stages to maintain coordination while preserving flexibility as priorities evolve.

The Kanban workflow includes the following stages, as illustrated in Figure 1.4:

- **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.

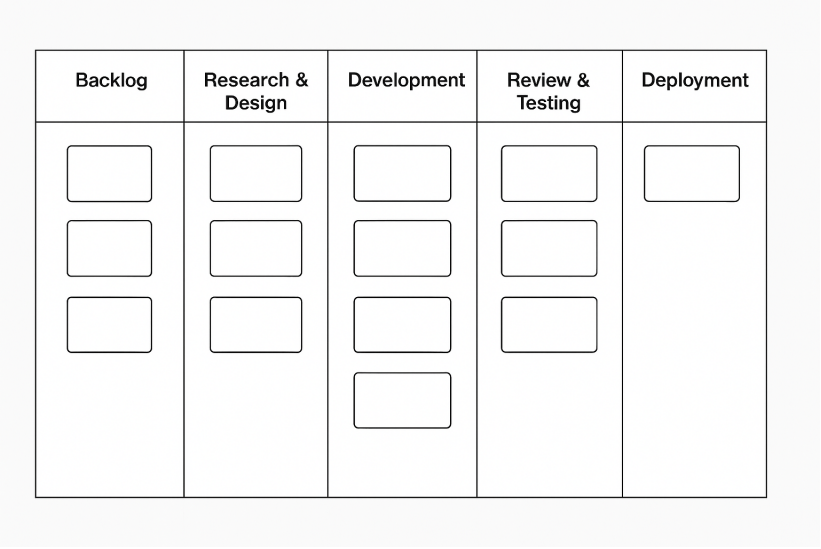

Figure 1.4 – Kanban Workflow

These columns represent flow states rather than rigid phases; items advance when explicit policies and Definitions of Done are satisfied, and may move back if additional work is discovered. The emphasis is on continuous flow, small batch sizes, and shortening cycle time.

1.3.3 Development Process

The project follows an iterative, incremental engineering cycle aligned with Agile and Kanban principles, designed to balance lightweight planning with continuous delivery, as illustrated in Figure 1.5. Planning and prioritization are continuous, batches are small, and delivery is flow-oriented. This cycle guides the work through structured stages, supported by dedicated tools and regular collaboration:

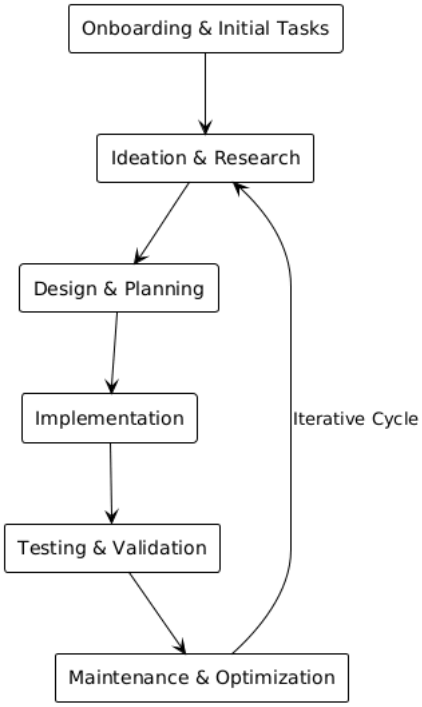

Figure 1.5 – Project Development Cycle

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

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

- **Ideation and Research:** Following onboarding, the project enters an exploration phase to clarify objectives, gather requirements, and investigate potential solution directions. Independent research complements collaborative discussions to assess feasibility and align priorities.
- **Design and Planning:** A lightweight, evolving design document presents technical choices, architectural considerations, and the proposed workflow. The document is refined iteratively through engineer reviews, and work items are tracked on the Kanban board for prioritization and flow.
- **Implementation:** Development is performed as small, vertical slices pulled from the board, respecting WIP limits, within the internal development environment and company-wide repository, with frequent integration and clearly scoped changes.
- **Testing and Validation:** Each change is accompanied by unit tests and validated through manual and AI-assisted code reviews. Functionality and reliability are verified continuously: automated unit tests ensure component-level correctness, while integration reviews and structured evaluations validate behavior within the larger system.
- **Maintenance and Optimization:** Refactoring, bug fixes, and updates are performed throughout development, particularly as dependencies evolve or methods become deprecated. This ensures that the solution remains consistent, maintainable, and aligned with evolving standards.

Collaboration is 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 provides timely feedback, clear guidance, and alignment on shared dependencies throughout the iterative cycle.

1.3.4 Project Timeline

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

Figure 1.6 shows the detailed project timeline.

Figure 1.6 – Project Timeline - Detailed Schedule

Conclusion

This chapter established the organizational setting and scope of the project, clarified the problem and objectives, and presented the methodology and flow-based workflow that will structure execution, along with an indicative timeline. The next chapter consolidates the conceptual foundations and situates the work within related efforts.

Chapter II

Business understanding and project requirements

Introduction

This chapter establishes the theoretical and contextual foundation of the project. It begins with an overview of AI technologies, highlighting their relevance to modern software engineering. Next, it presents the state of the art and the existing solutions for code quality and best practices. Finally, it defines the project requirements, showing how this work addresses gaps by integrating AI-driven assistance into key development phases.

1 Business Understanding

2.1.1 AI Foundations

2.1.1.1 AI Definition

Artificial Intelligence (AI) refers to computational systems capable of performing tasks that typically require human intelligence, including reasoning, learning, problem-solving, and decision-making [**?**]. AI encompasses a wide range of techniques with distinct capabilities and applications. Figure 2.1 illustrates a hierarchy of AI technologies.

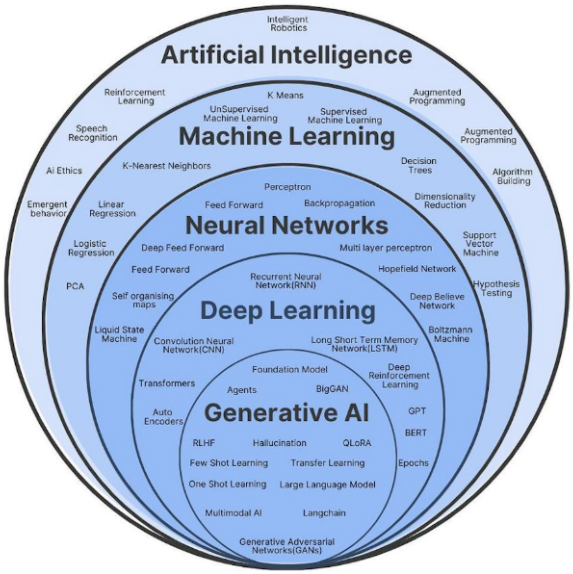

Figure 2.1 – Hierarchy of AI Technologies (adapted from [**?**])

Key categories include:

- **Symbolic AI:** Rule-based reasoning and knowledge representation systems.
- **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 [**?**].

2.1.1.2 Large Language Models (LLMs)

LLMs represent a breakthrough in generative AI, trained on extensive natural language and code corpora [**?**]. Unlike traditional tools, LLMs understand context, semantics, and intent, enabling complex reasoning across domains. Figure 2.2 provides a chronological overview of LLM development from 2018–2024.

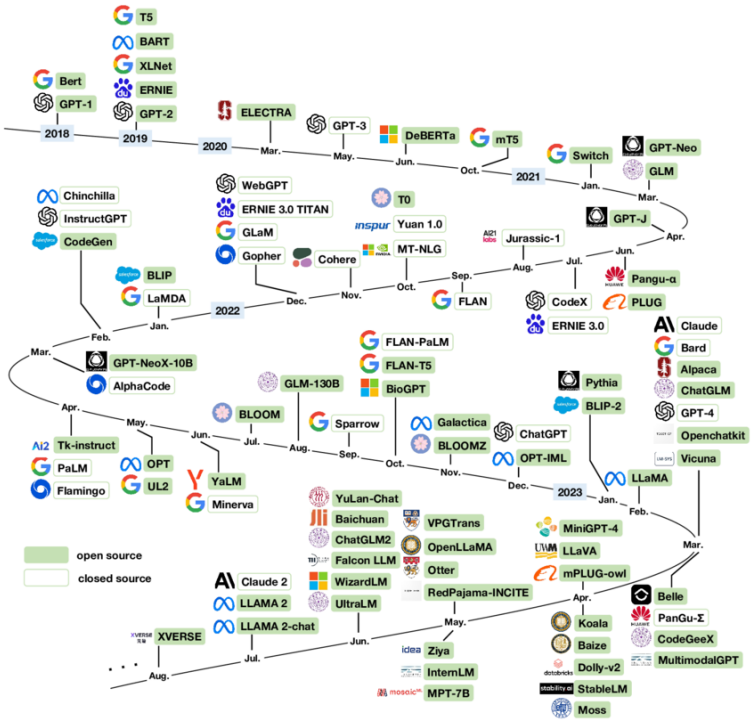

Figure 2.2 – Chronological Overview of Large Language Models (LLMs) (adapted from [**?**])

LLM capabilities include:

- **Natural language understanding:** interpret instructions and context in plain language.
- **Pattern recognition:** spot patterns and anomalies across text and data.
- **Content generation:** produce coherent text, summaries, and structured outputs on demand.
- **Reasoning and inference:** connect facts and constraints to propose solutions.

In software development, these capabilities translate into:

- **Contextual code analysis:** surface issues with awareness of surrounding code and intent.
- **Intelligent code generation:** draft functions and refactors that match local patterns.
- **Explanatory documentation:** explain changes and APIs in concise, developer-friendly language.

- **Semantic standards enforcement:** uphold internal conventions and architectures beyond style.

2.1.1.3 AI Agents

An agent is "a computer system that is situated in some environment and that is capable of autonomous action in this environment in order to meet its design objectives" [**?**]. In our context, AI agents are LLM-powered systems that not only generate text but also take actions toward a goal. They plan, call tools, and learn from prior steps to complete multi-stage tasks more reliably than a standalone LLM. In engineering settings, agents orchestrate LLM capabilities within workflows, integrating with IDEs, testing frameworks, repositories, and CI systems to deliver end-to-end assistance.

Why build with agents? Standalone LLMs excel at focused tasks (translation, drafting emails) but fall short on complex endeavors that demand planning, external data, and iterative decisions. Agents close this gap by combining LLM reasoning with memory, structured planning, and tool use so they can operate over real-world systems and fresh information.

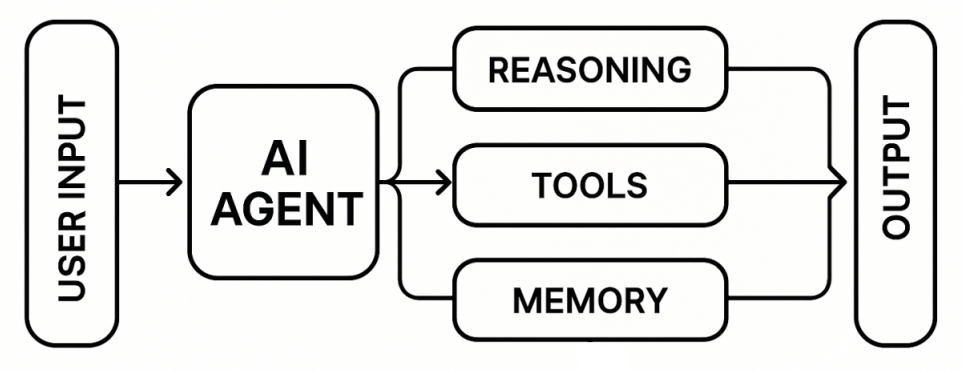

Figure 2.3 – AI Agent Architecture (adapted from [**?**])

A well-designed agent typically follows the layered architecture shown in Figure 2.3:

- **Input handling:** capture and preprocess user input.
- **Core reasoning (LLM):** use a prompt plus context to decide the next step.
- **Action layer:** integrate with tools (APIs, scripts, databases).
- **Memory layer:** store past interactions or retrieved data.
- **Output formatting:** return structured or human-readable results.

Despite their strengths, AI agents carry practical constraints: computational cost (model inference and tool invocations), context-window limits that can drop relevant history on long tasks, accuracy variability under domain shift, and integration complexity around security, permissions, and monitoring when calling external tools and data. In practice, these risks are reduced with scoped prompts and guardrails, retrieval-augmented grounding, human-in-the-loop checkpoints for high-impact actions, and operational controls over latency and spend.

2.1.2 The AI Revolution in Software Engineering: AI-Enhanced SDLC

Modern software development faces increasing complexity. Traditional practices often struggle to maintain code quality while meeting deadlines. AI technologies address this by embedding intelligent assistance directly into the development workflow.

Industry adoption highlights this impact: surveys show AI-generated code accounts for a growing portion of development output in major organizations [**? ?**]. AI integration addresses critical challenges such as maintaining code quality, reducing technical debt, and scaling development practices across teams.

Concretely, AI assistance spans the Software Development Life Cycle (SDLC) as follows:

- **Requirements and Planning:** Estimate timelines, identify ambiguities, and translate requirements into technical specifications.
- **Design and Architecture:** Recommend patterns, detect anti-patterns, and ensure compliance with organizational standards.
- **Implementation:** Context-aware code completion, best practice enforcement, bug detection, and refactoring guidance.
- **Testing and Quality Assurance:** Generate test cases, identify edge cases, and prioritize test execution.
- **Deployment and Maintenance:** Monitor performance, predict issues, and recommend optimizations.

As a result, development practices shift:

- **From Reactive to Proactive:** Feedback is delivered continuously during coding and design, preventing issues before they require refactoring.

- **From Inconsistent to Scalable:** AI agents provide uniform, expert-level guidance across teams and codebases.
- **From Static to Adaptive:** AI adapts to project-specific patterns, team preferences, and evolving best practices.
- **From Isolated to Integrated:** Guidance is embedded within workflows rather than treated as a separate phase, bridging planning, implementation, testing, and maintenance.

Overall, the AI-enhanced SDLC moves teams from reactive, fragmented practices to a proactive, adaptive, integrated approach. This aligns with our goal of delivering real-time, framework-specific guidance inside the IDE to improve developer efficiency and maintain code quality.

2 State of the Art and Existing Solutions

Building on the foundations above, this section examines both market-available solutions and environment-specific approaches to understand the current landscape of code quality enforcement and developer assistance, identifying gaps that the proposed solution aims to fill.

2.2.1 State of the Art

The software development market offers a variety of AI-powered tools and platforms designed to enhance code quality and developer productivity. These solutions represent the current state of the art in intelligent development assistance. To orient this landscape, we group offerings into the following categories:

- **AI Code Assistants:** GitHub Copilot, Amazon CodeWhisperer, and Tabnine provide AI-powered code completion and generation, helping developers write code more efficiently.
- **AI-Powered IDEs:** Tools 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, offering automated suggestions and quality assessments.

Common capabilities include:

- **General Code Analysis:** Broad pattern recognition and quality assessment across multiple languages and frameworks.
- **AI-Powered Suggestions:** Recommendations for code improvements, refactoring, and general best practices.
- **IDE Integration:** Seamless integration with widely used environments such as VS Code, IntelliJ, and Eclipse.

2.2.2 Existing Environment-Specific Approaches

Within our development environment, software engineers rely on a combination of modern AI-powered tools and traditional mechanisms to maintain code quality.

- **Code Reviews:** Human reviewers provide context-aware feedback on design quality, readability, maintainability, and adherence to standards. Feedback is high-level but often delayed and resource-intensive.
- **Presubmit Checks:** Automated scripts enforce style guides, compilation correctness, and basic safety constraints. They are fast but primarily focus on surface-level checks.
- **Coding Assistant:** Internal IDE features AI-powered code completion, generation, and basic suggestions.
- **Linters:** IDE-integrated linters provide real-time style and deprecation warnings but do not enforce complex best practices.
- **Rule-Based Checks:** Enforce coding conventions and naming schemes consistently but cannot reason about complex or context-dependent practices.

2.2.3 Gap Analysis and Opportunity

2.2.3.1 Gaps in current approaches

While both market solutions and environment-specific approaches provide valuable capabilities, they leave significant gaps in enforcing internal framework-specific best practices during the coding phase.

Limitations of Market Solutions

- **Generic analysis scope:** broad patterns across languages and frameworks, but limited grasp of internal conventions and architectural intent.
- **External dependency and data handling:** reliance on third-party services can raise confidentiality, residency, and compliance concerns for internal code.
- **Customization overhead:** difficult to encode and maintain project-specific practices and evolving architectural patterns.

Limitations of Environment Solutions Figure 2.4 illustrates how current mechanisms provide feedback at different points in the workflow, highlighting the gap during active coding.

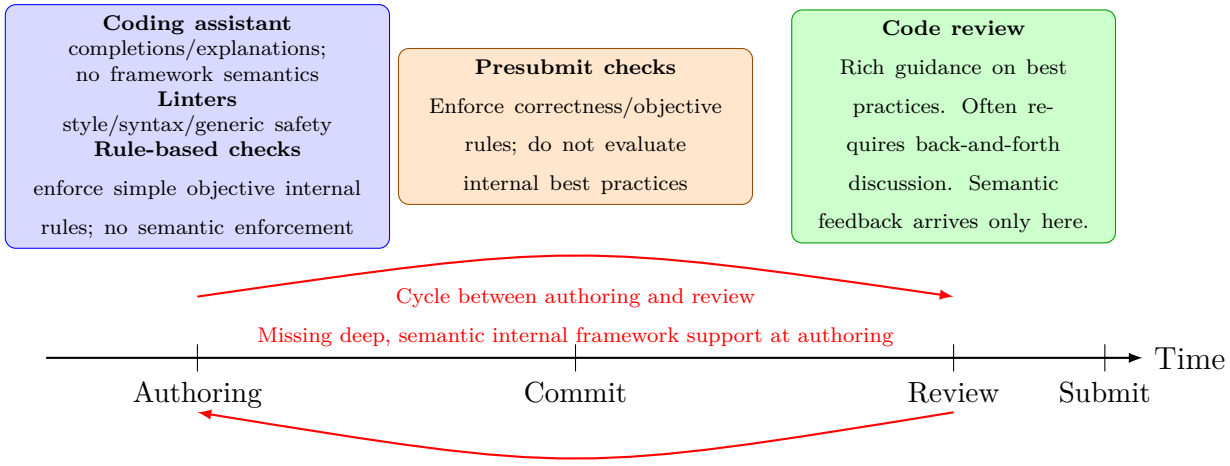

Figure 2.4 – Current feedback across the development timeline.

As the timeline suggests, authoring tools provide helpful but limited coverage: **coding assistants and linters** improve style, syntax, and generic safety, and **rule-based checks** can enforce simple, objective internal conventions (e.g., naming, imports). However, **none of these provide semantic enforcement of internal framework best practices at the point of writing**. At commit time, **presubmit checks** primarily ensure correctness rather than best-practice conformance. The first truly **semantic, framework-aware feedback** typically appears in **code review**, where it is rich but time-bounded and variable—often prompting back-and-forth and rework.

2.2.3.2 Impact of Delayed Feedback

The current workflow introduces several challenges:

- **Inconsistent Quality Standards:** Lack of real-time guidance leads to variable adherence across teams.
- **Prolonged Review Cycles:** Multiple iterations are required to fix issues discovered late.
- **Technical Debt Accumulation:** Delayed feedback leads to inconsistencies and long-term maintenance costs.
- **Developer Frustration:** Repetitive corrections decrease productivity.
- **Increased Costs:** Late discovery of issues exponentially increases remediation effort.

2.2.3.3 Opportunity for Framework-Specific AI Solutions

The analysis reveals a clear opportunity for integrating framework-specific AI solutions. Combining the intelligence of market tools with the specificity required for internal frameworks, these solutions can deliver real-time, context-aware guidance directly in the coding phase.

Such a solution addresses the observed gap:

- Real-time adherence to internal framework best practices.
- Integration directly into the developer workflow.
- Proactive feedback that reduces review effort and technical debt.

This motivates the development of an LLM-powered assistant integrated into the coding phase within the **Integrated Development Environment (IDE)**. The IDE concentrates the artifacts and signals of software work (source files, diagnostics, version state, and developer intent), so in-editor guidance shortens the loop between authoring and feedback and keeps advice anchored in local context.

3 Project Requirements

We now translate the identified opportunity into concrete requirements. The proposed solution integrates LLM-powered assistance directly into the developer workflow within the IDE to provide real-time, actionable guidance while maintaining performance, usability, and scalability.

2.3.1 Use Case Analysis

Understanding how developers will interact with the system is critical for designing effective functionality. Use case analysis provides a structured approach to capture user-system interactions, identify key actors, and define the boundaries of the system. This ensures that the system delivers targeted support precisely where it is needed during the active coding phase.

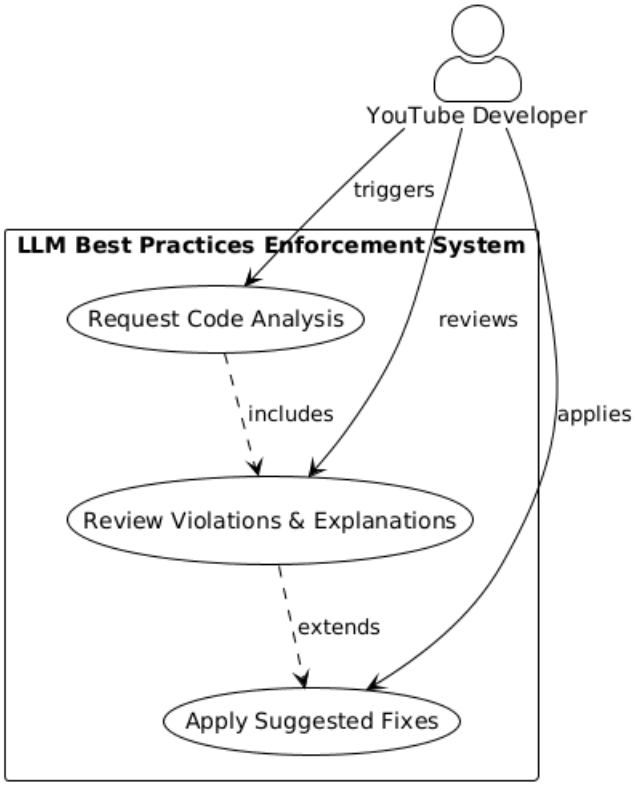

Figure 2.5 – System Use Case Diagram

Figure 2.5 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:

1. **Request Code Analysis:** Trigger real-time evaluation of code for adherence to framework best practices.
2. **Review Violations and Explanations:** Inspect flagged violations, with clear, context-specific explanations provided by the system.
3. **Apply Suggested Fixes (Optional):** Accept, modify, or reject actionable AI-generated suggestions to improve code quality.

This design ensures that developers remain in control of their workflow while benefiting from comprehensive, intelligent feedback. Optional scenarios, such as applying fixes, allow flexibility and respect developer autonomy. The use case diagram focuses on core interactions, leaving authentication and configuration workflows out of scope.

2.3.2 Functional Requirements

Based on the use cases and the gaps identified in existing tools, the system must provide the following core functionalities:

- **Detect Framework Violations:** Identify violations of internal YouTube framework best practices in real-time.
- **Provide Contextual Explanations:** Deliver developer-friendly explanations that clarify why a particular pattern is problematic.
- **Generate Actionable Fixes:** Suggest concrete AI-driven solutions aligned with framework conventions.
- **Enable Developer Interaction:** Allow developers to accept, reject, or modify suggested fixes, maintaining workflow control.
- **Maintain Contextual Relevance:** Ensure feedback remains accurate and properly anchored as code evolves.
- **Seamless IDE Integration:** Embed within the existing internal IDE without disrupting normal development processes.

2.3.3 Non-Functional Requirements

In addition to core functionality, the system must satisfy broader quality criteria:

- **Performance:** Provide near real-time feedback to avoid interrupting workflow.
- **Scalability:** Efficiently handle large codebases and multiple simultaneous users.
- **Maintainability:** Make it easy to modify, refactor, and repair existing components (rules, prompts, pipelines) with clear abstractions, tests, and documentation.
- **Reliability:** Operate robustly in production with minimal downtime.

- **Security and Privacy:** Comply with organizational policies, safeguarding code and data.
- **Usability:** Deliver concise, context-aware, and minimally intrusive feedback.
- **Extensibility:** Allow adding new capabilities (rules, models, tools, IDE touchpoints) via well-defined extension points without changing existing components.

Table 2.1 summarizes functional and non-functional requirements.

Table 2.1 – Summary of Project Requirements

Requirement Type	**Description**
Functional	Detection, explanations, fixes, interaction, context, IDE integration
Non-Functional	Performance, scalability, maintainability, reliability, security, usability, extensibility

These requirements directly address the gaps identified in both market and environment-specific solutions. By embedding proactive, context-aware feedback into the coding workflow, the system:

- Reduces framework-specific errors during development.
- Improves adherence to YouTube internal standards.
- Enhances developer productivity by providing guidance in real-time.

Conclusion

This chapter set the foundations: AI, LLMs, and agents; why agents are needed; and why the IDE is the right place for real-time guidance. It mapped AI's role across the SDLC, reviewed market and environment approaches, and showed a gap in enforcing internal framework best practices at authoring. From this analysis, it defined the opportunity and the project's requirements. The next chapter presents the system design and architecture that realizes authoring-time, framework-aware guidance in the IDE.

Chapter III

System Design and Architecture

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 Chapter 2 , it details 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

3.1.1 High-Level Architecture

The system architecture is designed to integrate seamlessly into the developer's existing workflow while providing intelligent, context-aware feedback. It consists of two main components:

- **IDE:** The developer's workspace that hosts the YouTube IDE Extension, which serves as the entry point and user-facing interface. It's a familiar part of YouTube developers' workflow that aims to improve different aspects of the developer experience.
- **AI Agent Framework:** This is the serving infrastructure for LLM-powered applications, or agents developed by YT DevInfra. It aims to make deploying such applications easy by providing out-of-the-box functionality and encouraging reuse of existing LLM libraries. It hosts the LLM Best Practices Agent, which performs code analysis and generates best practice suggestions.

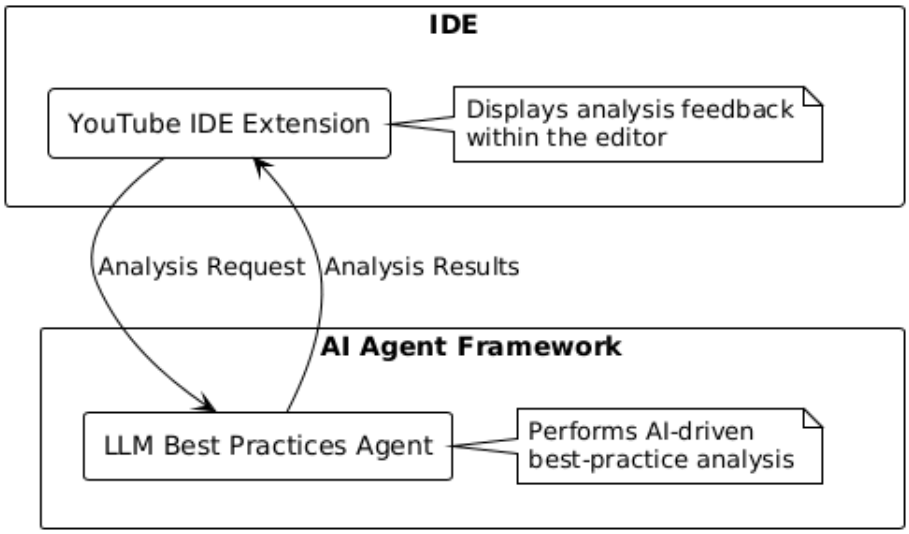

Figure 3.1 – High-Level System Architecture

Figure 3.1 illustrates the separation between the IDE extension and the AI processing backend. The IDE provides immediate access to analysis capabilities, while the AI Agent Framework handles computationally intensive tasks. This separation allows independent scaling of AI capabilities without impacting IDE responsiveness.

3.1.2 System Workflow

The system operates through a streamlined workflow beginning when a developer triggers analysis via the IDE Extension (Figure 3.2).

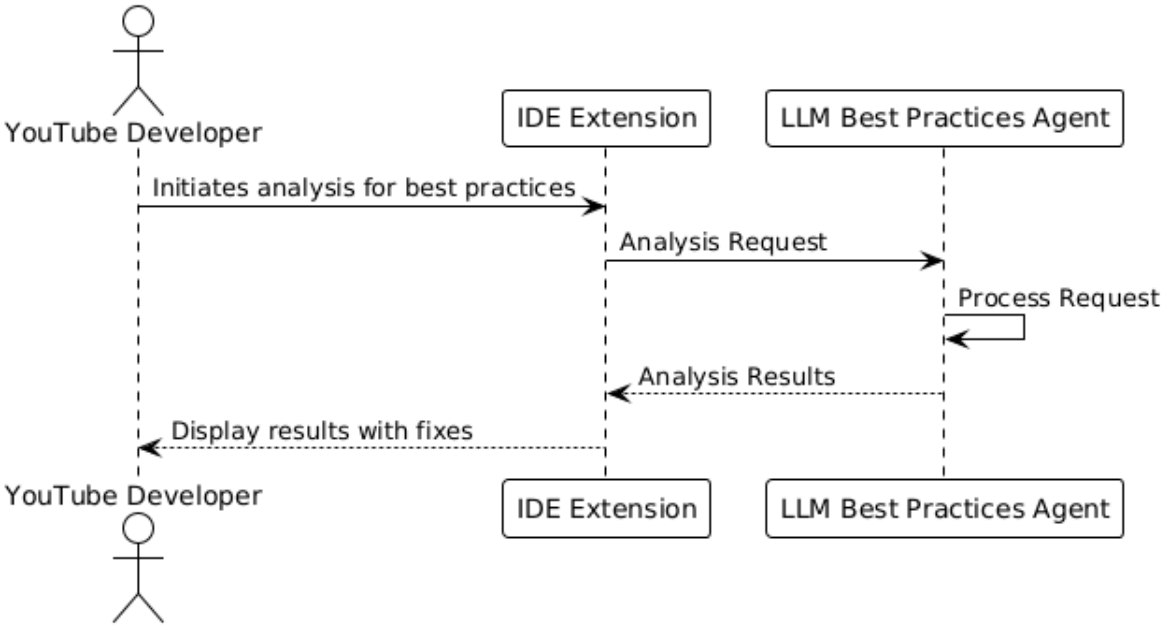

Figure 3.2 – System Interaction Sequence Diagram

The primary workflow includes:

1. **Analysis Initiation:** The developer triggers analysis on the open file.
2. **Request Submission:** The IDE Extension submits the request to the AI agent, identifying the target file.
3. **Processing:** The agent performs best-practices analysis.
4. **Result Delivery:** The agent returns structured findings.
5. **Presentation:** The IDE Extension displays violations and suggested fixes in the editor.

This workflow decouples the IDE from heavy computation, maintaining responsiveness while delegating analysis to the specialized agent framework.

2 LLM Best Practices Agent

The LLM Best Practices Agent is the core intelligence engine of the system. Its responsibilities include analyzing code, detecting violations of internal YouTube framework best practices, providing human-readable explanations, and suggesting actionable fixes. This component represents the intersection of AI capabilities with domain-specific software engineering expertise.

3.2.1 Core Tools Architecture

We begin with the building blocks the agent orchestrates. Aligned with the AI agent architecture introduced in Chapter , the agent uses specialized tools, each responsible for a specific step of the best practices enforcement pipeline:

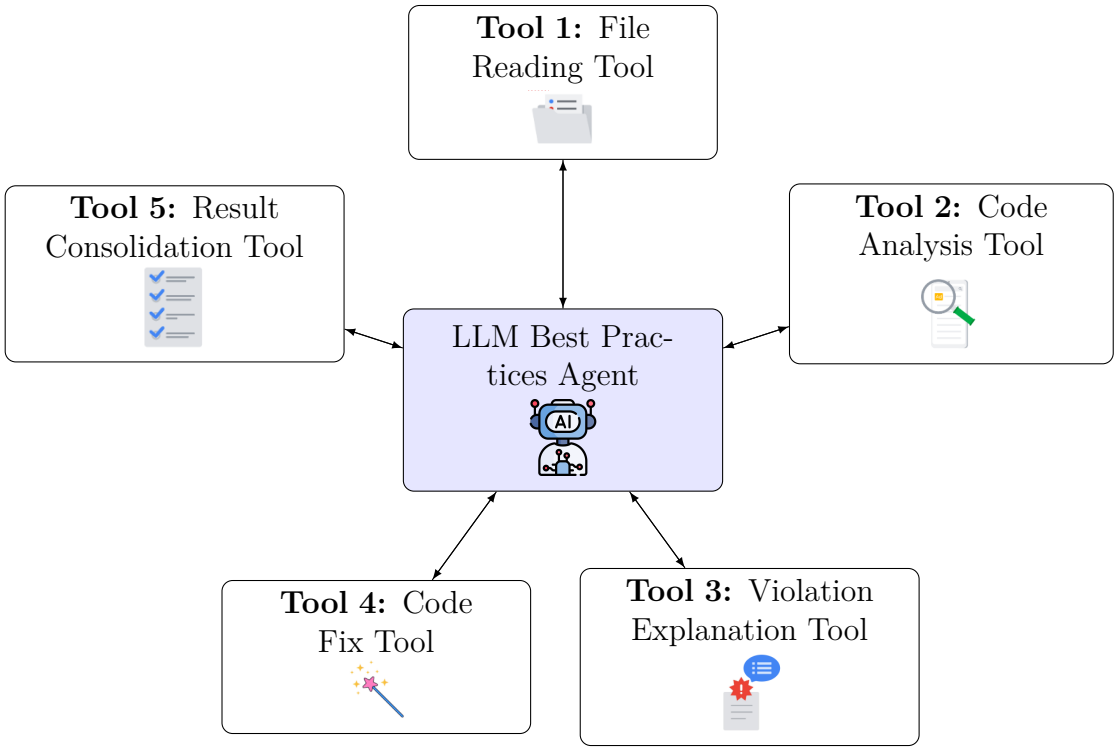

Figure 3.3 – The LLM Best Practices Agent orchestrating interactions with its specialized tools

As illustrated in Figure 3.3, the agent sits at the center and interacts with five specialized tools to complete the analysis. The roles of these tools are summarized below:

- **File Reading Tool:** Retrieves complete file content. Ensures full context for downstream analysis.
- **Code Analysis Tool:** Performs structural and semantic analysis against internal framework rules, considering local context around each finding. Combines static analysis heuristics with LLM-based reasoning to flag location, rule identifier, and a short rationale.
- **Violation Explanation Tool:** Converts raw violations into human-readable explanations, providing actionable insight and contextual rationale for developers.

- **Code Fix Tool:** Suggests concrete, minimal edits that align with framework conventions. Generates patch-style suggestions or structured steps to help developers resolve the violation confidently.
- **Result Consolidation Tool:** Aggregates outputs from all tools into a structured response consumable by the IDE extension, ensuring clear and contextually accurate presentation.

3.2.2 Integration with LLM Infrastructure

With tool responsibilities defined, the agent interfaces with an internal AI platform hosting multiple LLM models. Key design considerations include:

- **Model-Agnostic Orchestration:** The architecture supports seamless adoption of new models without modifying the agent workflow.
- **Stable Interfaces:** Each tool interacts with the LLM via encapsulated, stable APIs to ensure maintainability.

3.2.3 Agent Architecture Options

Now that the tools are defined, the question is how the agent will orchestrate them. We consider two agent architecture options our internal AI Agent Framework offers: Executable and ReAct. We explored these paradigms for the agent, drawing inspiration from established AI agent patterns [**?**]. Figure 3.4 illustrates the fundamental differences in control flow between these two options.

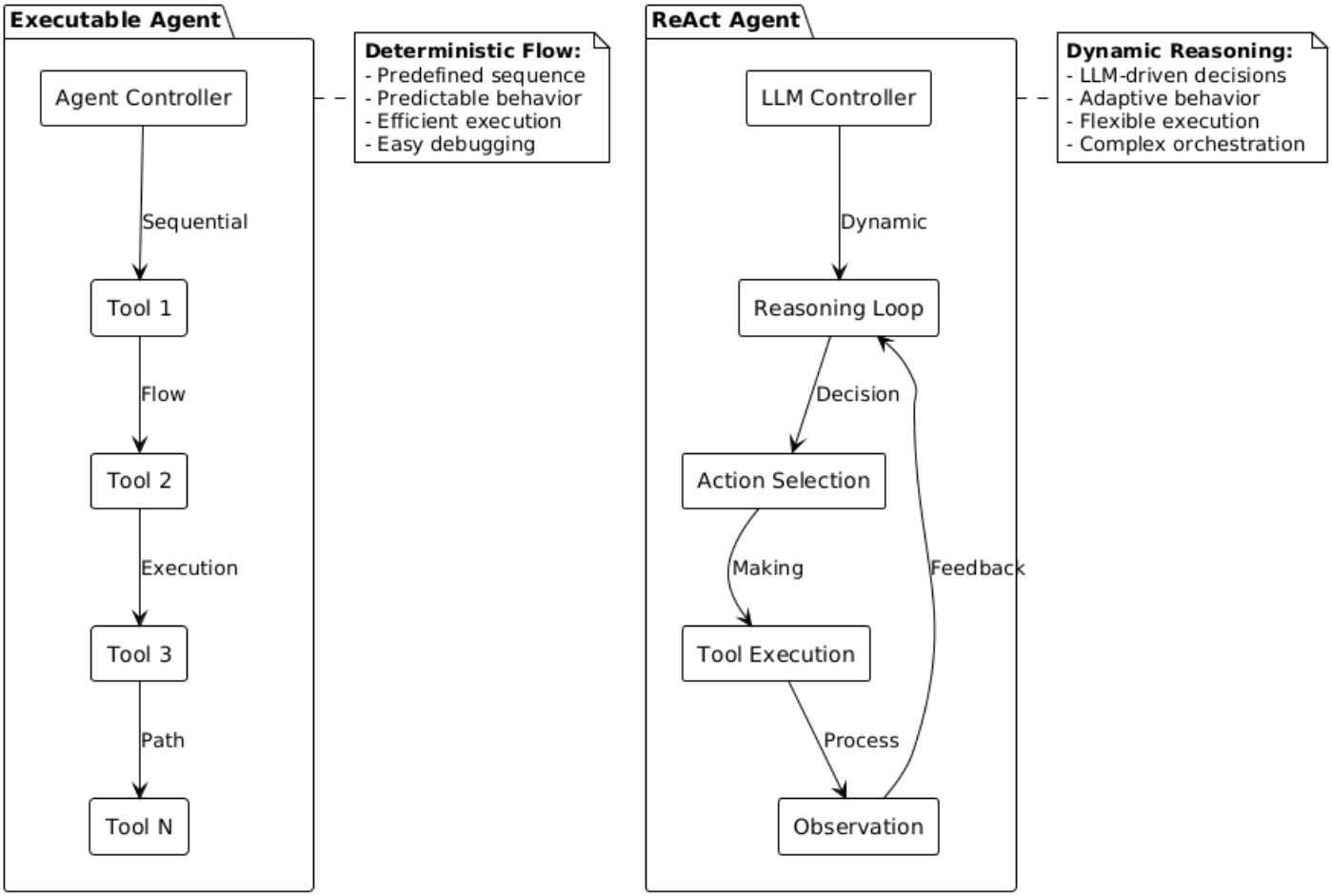

Figure 3.4 – Comparison of Executable Agent vs. ReAct Agent Architectures

- **ReAct Agent:** Integrates *reasoning and acting* in a closed loop. The agent interprets the goal, plans the next step, selects and invokes a tool, observes the result, and updates its internal trace before continuing. This iterative control flow supports open-ended problem solving, dynamic tool chaining, and exploratory analysis. ReAct agents typically maintain a short-term action log (thoughts, tool calls, observations) that explains how outcomes were reached and can be surfaced as an execution trace for review.

- **Executable Agent:** Encodes a deterministic, predefined workflow as an explicit sequence or graph of steps. Each step is a well-typed tool or validator with clearly defined inputs, outputs, and pre/post-conditions, and the LLM is invoked within steps for domain-specific processing rather than global orchestration. This structure enables precise routing, step-level retries and fallbacks, policy gates, and comprehensive telemetry at each stage, making execution easy to audit, measure, and operate at scale.

3.2.4 Processing Strategy Options

Complementing these architecture options, the processing strategy governs how the agent sequences and (when appropriate) parallelizes analysis across findings. It directly impacts IDE responsiveness, determinism, and operational cost. In our setting, a single file may surface many candidate violations; to handle this efficiently and predictably, we considered three strategies:

- **Fully Sequential:** Processes one finding at a time in a clear, step-by-step flow. Each finding is analyzed, optionally fixed, and then the next one is handled. This makes the execution easy to follow and the output straightforward to review.
- **Fully Parallel:** Processes all findings at once. The agent analyzes each finding independently and then returns a single, consolidated result. This mode is suited to fast, whole-file scans and bulk reporting.
- **Hybrid Approach:** Processes a small set of findings together, then moves on to the next set. It combines the readability of sequential steps with the speed of parallel checks, keeping the experience smooth for larger files.

3 Evaluation Framework for Architecture Decisions

Given the complexity and multiple viable options for agent architecture and processing strategies, empirical evaluation is essential. Decisions cannot be based solely on intuition; performance, reliability, and quality need to be measured under realistic conditions.

We consider three candidates:

- **Sequential Executable Agent**
- **Parallel Executable Agent**
- **ReAct Agent**

3.3.1 Test Suite Framework

We selected a set of 12 diverse source files. For each file, we manually reviewed the code to identify violations, then defined evaluation cases with expected outcomes: findings, explanations, and minimal fixes. This provides a reproducible ground truth and enables fair comparisons across runs.

Coverage dimensions

- **File size and structure**: small, medium, and large files.
- **Violation density**: none (clean baselines), sparse, dense, and clustered violations.
- **Pattern complexity**: straightforward single-line issues vs. multi-line, context-dependent patterns.
- **Error-inducing cases**: inputs designed to trigger errors or timeouts to validate failure handling and recovery.
- **Rule categories**: naming/structure, lifecycle/ordering, performance-sensitive conventions, and readability/documentation.
- **Conflict/overlap**: cases where multiple rules trigger on the same span or produce competing suggestions.
- **Language constructs**: different control-flow forms, data structures, and API usages.

3.3.2 LLM-as-a-Judge Methodology

Each evaluation case specifies an expected output (gold findings, explanations, and minimal fixes). Direct string comparison is brittle: semantically correct answers may differ in phrasing, ordering, or formatting. To compare the agent output to the gold robustly, we use an independent ***LLM-as-a-Judge*** that evaluates semantic equivalence and quality.

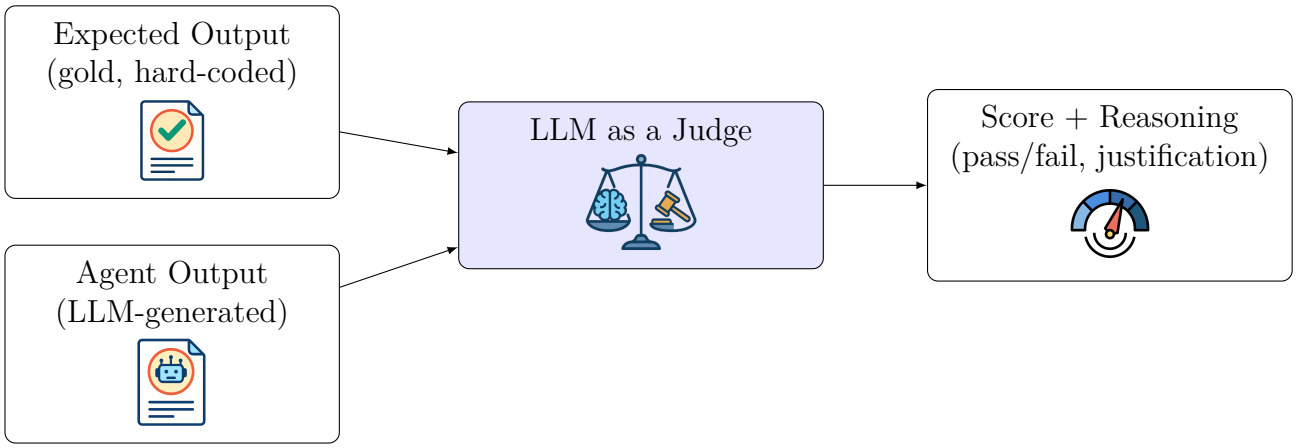

Figure 3.5 – LLM-as-a-Judge

LLM-as-a-Judge is an evaluation paradigm in which a separate LLM grades model outputs against a reference (gold) or via pairwise comparison using an explicit rubric, then returns a

score and brief rationale; prior work reports strong correlation with expert human judgments when prompts and rubrics are well specified [**?**].

In our evaluation, we use a **rubric-based, reference comparison**. "Rubric-based" means the judge applies a small, explicit checklist of acceptance criteria; "reference comparison" means the agent output is evaluated against a hand-labeled gold reference rather than another model output. The judge compares the agent output to the gold reference and returns a pass/fail decision with a brief rationale.

Rubric (acceptance criteria):

- Missing any expected convention → fail
- Hallucinated/extra conventions not in gold → fail
- Ordering differences do not matter
- Values/ranges must match within a small tolerance (e.g., ±10%)
- Returned an error for a case where a valid answer exists → fail
- Explanations and suggested fixes must be consistent with the referenced convention and not contradict gold
- Formatting differences are ignored unless they change meaning

A case passes only if all rubric criteria are satisfied; any unmet criterion results in a fail.

3.3.3 Key Metrics

The evaluation framework measures three primary dimensions:

- **Accuracy:** Semantic correctness of analysis results and suggested fixes, as judged by the LLM evaluator.
- **Latency:** End-to-end response time, ensuring real-time usability in the IDE.
- **Cost:** Token consumption, API usage, and compute resource requirements.

Each agent architecture option is assessed using the test suite and the LLM-as-a-Judge methodology, producing data-driven insights into trade-offs between determinism, throughput, accuracy, and operational cost. This combination of structured evaluation and modular design ensures that architecture and processing strategies can be selected with confidence, balancing performance, reliability, maintainability, and developer usability.

4 YouTube IDE Extension Integration

As introduced above, the YouTube IDE Extension is the user-facing interface that integrates analysis into developers' daily workflow. This section details its architecture and interaction patterns, focusing on in-editor, context-aware feedback while preserving IDE responsiveness.

3.4.1 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.

3.4.2 User-Triggered vs. Automatic Analysis

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 3.1 summarizes the comparison of user-triggered vs. automatic analysis approaches:

Table 3.1 – 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.

3.4.3 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.

3.4.4 User Interaction Flow

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

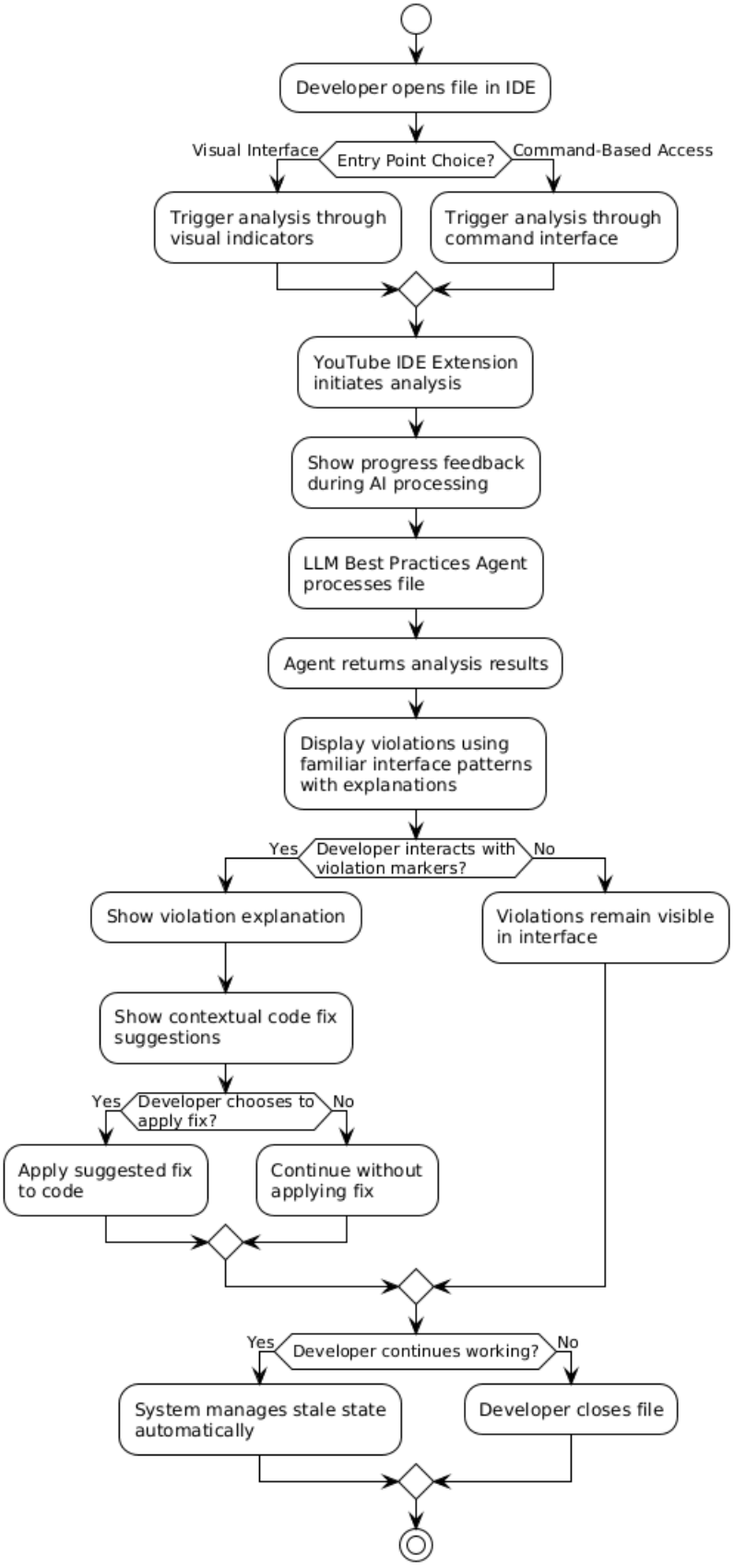

Figure 3.6 – YouTube IDE Extension User Interaction Flow: Complete developer journey from analysis trigger to fix application

As illustrated in Figure 3.6, 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.

5 Data Models and Interfaces

3.5.1 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.

3.5.2 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 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 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

This chapter defined the end-to-end design of the system: a high-level architecture that separates the IDE extension from the AI backend; the LLM Best Practices Agent with its core tools and model integration; alternative agent architectures and processing strategies; a pragmatic evaluation framework leveraging a rubric-based LLM-as-a-Judge; and the IDE integration and data contracts that make the experience usable in practice. Together, these elements provide a clear, modular foundation for enforcing framework best practices with AI, while enabling data-driven selection among viable execution options. The next chapter turns to implementation details and empirical results based on this design.

Chapter IV

Implementation

Introduction

Building on the system design established in Chapter 3, this chapter details the implementation of the system. We begin with the working environment and technology stack, then present the concrete implementation of the AI agent backend and the IDE extension frontend.

1 Working Environment

4.1.1 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.

4.1.1.1 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.

4.1.1.2 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.

4.1.1.3 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 [**?** **?**]. It was part of the development environment rather than the deployed technology stack.

4.1.1.4 Internal Repository Integration

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

4.1.2 Project Management and Documentation

4.1.2.1 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 [**?**].

4.1.2.2 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 [**?**].

4.1.2.3 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 [**?**].

4.1.2.4 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) and internal platforms operated within Google (e.g., YouTube DevInfra Agent Framework, internal AI platform).

4.2.1 Backend Technologies (AI Agent)

4.2.1.1 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 [**?**]. The language's dynamic typing and support for artificial intelligence and machine learning libraries make it suitable for AI agent development [**?**].

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.

4.2.1.2 YouTube DevInfra Agent Framework

The implementation utilizes the YouTube DevInfra Agent Framework which is the serving infrastructure that underpins all YouTube agents, providing standardized execution, deployment,

and operational primitives for LLM-powered applications. This framework natively 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.

4.2.1.3 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.

4.2.1.4 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.
- **Prompt Engineering Libraries**: Frameworks for constructing, optimizing, and managing prompts.

4.2.2 Frontend Technologies (IDE Extension)

4.2.2.1 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 [**?**]. 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.

4.2.2.2 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 [**?**]. 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.

3 Realization

The realization phase translates architectural decisions into a concrete implementation of the system. It consists of two primary components: (1) the **AI Agent backend**, responsible for code analysis and violation handling, and (2) the **IDE Extension frontend**, responsible for developer interaction. This section presents the detailed implementation of the agent and its integration with the IDE extension, with brief references to evaluation results where it clarifies the rationale behind specific choices.

4.3.1 Agent Realization

4.3.1.1 Evaluation Results and Architecture Decision

To determine the most suitable agent design, we evaluated three candidate architectures: the Parallel Executable, Sequential Executable, and ReAct Agent. These were assessed against 12 representative test cases covering a range of violation patterns and code complexities.

For a simple single-violation file, the results are shown in Table 4.1. Parallel outperformed Sequential by ∼33% in latency, while both were vastly more efficient than ReAct in token usage.

Table 4.1 – 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

Scaling to the full evaluation suite (Figures 4.1–4.3) revealed deeper trade-offs.

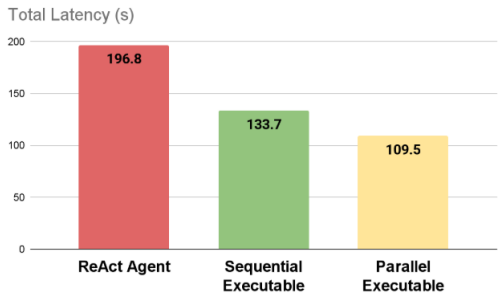

Figure 4.1 – Latency Performance Across 12 Test Cases

As shown in Figure 4.1, the latency profile favors the fully parallel executable agent, which completed the suite roughly 20–25% faster than the sequential counterpart. However, latency alone is not decisive for IDE scenarios, where reliability under load is equally critical.

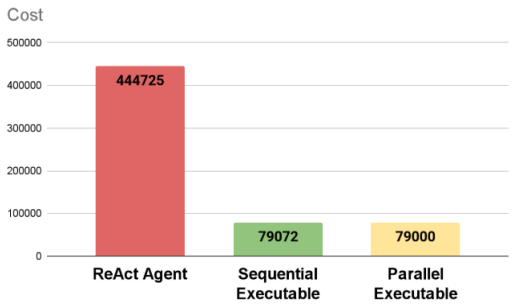

Figure 4.2 – Cost Analysis Across 12 Test Cases

As shown in Figure 4.2, token cost highlights a fundamental inefficiency in the ReAct agent: due to iterative reasoning–acting loops, it consumes orders of magnitude more tokens (6x in our suite) to achieve comparable outcomes. This renders ReAct non-viable for an interactive IDE use case.

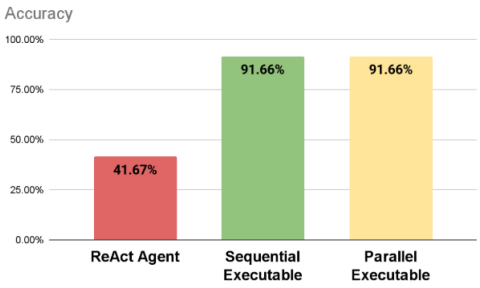

Figure 4.3 – Accuracy Performance Across 12 Test Cases

As shown in Figure 4.3, both executable variants achieved high accuracy (Sequential at

91.7%, Parallel matching on successful runs). Yet, the apparent parity masks a practical limitation: under heavier workloads, the fully parallel strategy occasionally failed with resource-limit and deadline errors. In an IDE, such instability outweighs raw speed advantages.

Synthesizing these findings, we adopted a **Parallel Executable architecture with bounded concurrency**. By enforcing semaphore-based limits on in-flight LLM calls, the design retains most of the latency advantage of full parallelism while eliminating overload-induced failures. This yields a deterministic, high-throughput, and reliable execution profile appropriate for production IDE integration.

4.3.1.2 Agent Implementation Overview

The AI Agent implements the chosen architecture using Python and integrates with Google's internal AI platform hosting Gemini-based LLMs. The design is class-based and modular, ensuring extensibility, observability, and testability. The agent exposes a single `execute()` entry point, orchestrating a pipeline of specialized tools that handle file reading, analysis, explanation generation, fix generation, and result consolidation.

Configuration and Initialization At startup, the agent performs several critical setup tasks:

- **Model Selection:** Configured to use the latest Gemini-based model trained on internal Google code.
- **Convention Loading:** Best practices are defined as in-memory Python objects and loaded at startup into a cache for constant-time access.
- **Tool Registration:** The five specialized tools are instantiated and registered into a deterministic workflow.

Implementation Structure The implementation follows a layered orchestration pattern:

- A central agent class orchestrates the pipeline via dependency-injected tools.
- Each tool is encapsulated as a class with clear `run()` contracts and typed inputs/outputs.
- Metrics and token usage are logged per tool, enabling fine-grained observability.
- Separation of concerns allows tools to be replaced or extended without modifying orchestration logic.

4.3.1.3 Core Tools Implementation

The agent implements five specialized tools, each corresponding to one stage of the analysis pipeline. Tools are designed as independent classes that expose a public `run()` method, which enforces a clear input/output contract and raises typed errors when failures occur. This contract-based design makes the system modular, testable, and resilient to partial failures.

ReadFileFromWorkspace Tool **Contract:** input = file path; output = file content (string). The ReadFileFromWorkspace tool is responsible for retrieving the contents of the developer's source file. Implementation details include:

- **File system access:** Uses Python's built-in file handling with UTF-8 as the default encoding, while detecting and recovering from alternative encodings.
- **Error handling:** Raises typed errors such as `FileNotFoundError`, `PermissionDeniedError`, and `EncodingError`. These errors are logged and surfaced to the developer in structured form.
- **Robustness:** Handles large files by streaming content, ensuring memory efficiency.

CodeAnalysisTool **Contract:** input = file content; output = list of base violations. This tool forms the analysis core of the system. It detects framework violations by orchestrating LLM queries enriched with contextual information.

- **Prompt construction:** Implements a template system with slots for file type, code snippet, and dynamically filtered convention definitions.
- **Context injection:** Selects relevant conventions from the cache and injects them into the prompt to guide the model.
- **Response parsing:** Uses strict schema validation with recovery mechanisms for malformed LLM responses.
- **Performance optimizations:** Employs caching of templates and convention definitions, reducing repeated token usage and lowering latency.

ViolationExplanationTool **Contract:** input = base violation; output = natural-language explanation. The ViolationExplanationTool generates educational explanations that help developers understand not only what the violation is, but why it matters.

- **Contextualization:** Pulls in violation metadata (rule violated, line number, code snippet) to ground explanations in concrete evidence.
- **Generation style:** Prompts the LLM to balance technical accuracy with readability, avoiding overly generic statements.
- **Implementation detail:** Each explanation is post-processed for clarity, removing redundant phrasing and enforcing concise output.

CodeFixTool Contract: input = violation + explanation; output = code fix (annotated snippet). This tool provides actionable, safe, and educational fixes.

- **Safety:** Fixes are constrained to local code changes, avoiding edits that break APIs or dependencies.
- **Self-containment:** Ensures that each fix can be applied without requiring modifications in other files.
- **Contextual adaptation:** Uses file content and violation metadata to adapt fixes to the surrounding code style.
- **Educational emphasis:** Each fix includes explanatory comments describing why the change is required.
- **Error handling:** Raises a `FixGenerationError` if the LLM output cannot be parsed or validated as compilable code.

Finish Tool Contract: input = violations + explanations + fixes; output = structured response for IDE extension. The Finish Tool acts as the consolidation component, producing a structured JSON response that the IDE extension can directly render.

- **Aggregation:** Combines violations, explanations, and fixes into a unified structure keyed by violation ID.
- **Deduplication:** Identifies overlapping or redundant violations and merges them to reduce noise.
- **Validation:** Ensures schema compliance so that the IDE can reliably parse and render results.

- **Formatting:** Applies consistent formatting (line numbers, code blocks, explanations) to improve readability in the UI.

Collectively, these tools form a deterministic pipeline coordinated by the agent's `execute()` method. Each tool adheres to explicit contracts, which improves modularity, observability (per-tool token usage is logged), and long-term maintainability.

4.3.1.4 Processing Workflow

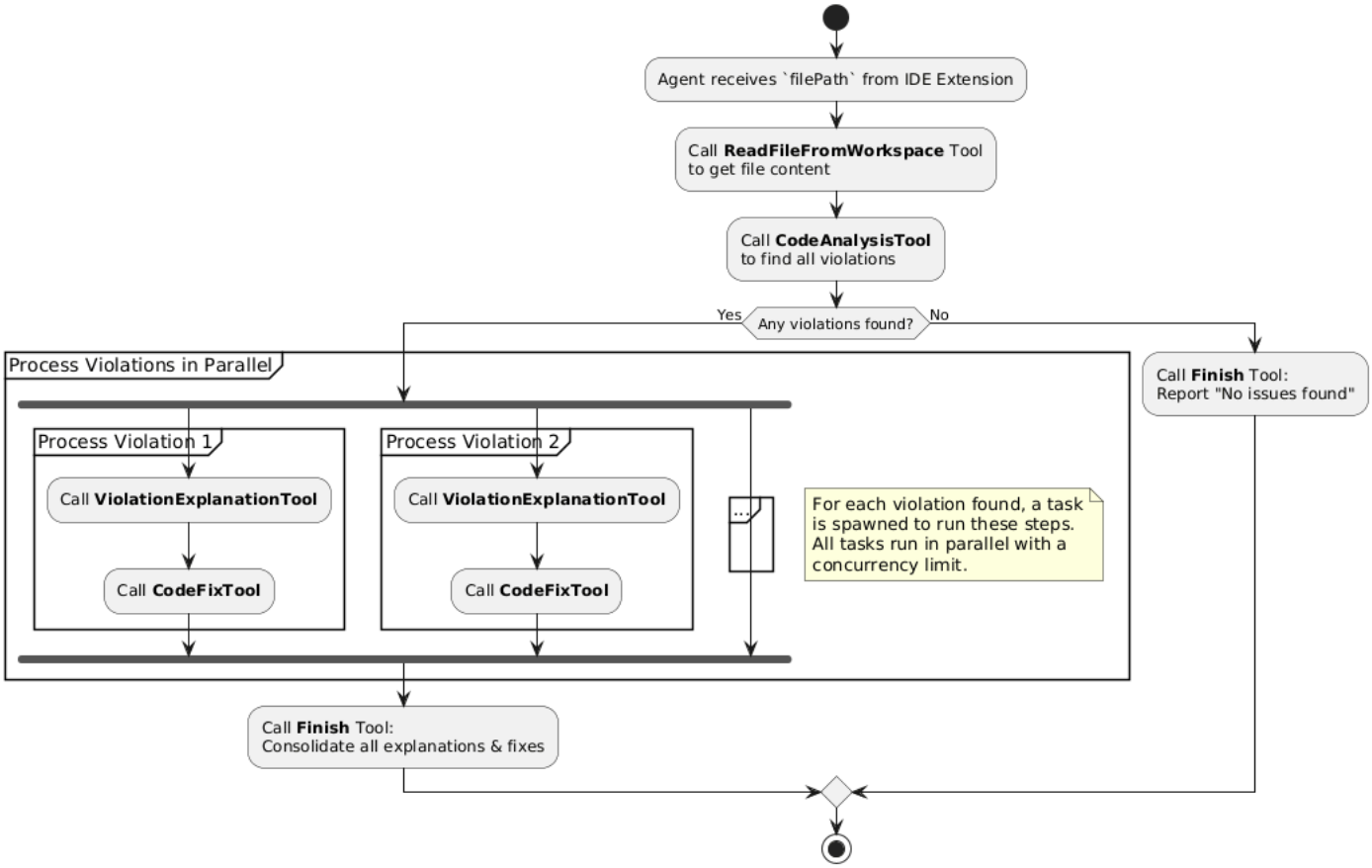

Figure 4.4 – Agent Processing Activity Diagram

As illustrated in the activity diagram in figure 4.4, the hybrid strategy is realized in three main stages:

1. **Sequential Preprocessing:** Full file content is read and all violations are identified.
2. **Parallel Processing:** Explanation and fix generation tasks are executed concurrently with semaphore-based limits on in-flight LLM calls.

3. **Result Consolidation:** Outputs are aggregated, validated, and prepared for the IDE extension.

4.3.1.5 Resilience and Error Handling

The system ensures graceful degradation under failures:

- Independent processing of violations isolates failures to individual tasks.
- Retry with exponential backoff mitigates transient network or LLM errors.
- Typed error classes facilitate debugging and developer support.
- Partial results are preserved, ensuring developers always receive usable feedback.

4.3.1.6 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.

4.3.2 Extension Integration

4.3.2.1 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.

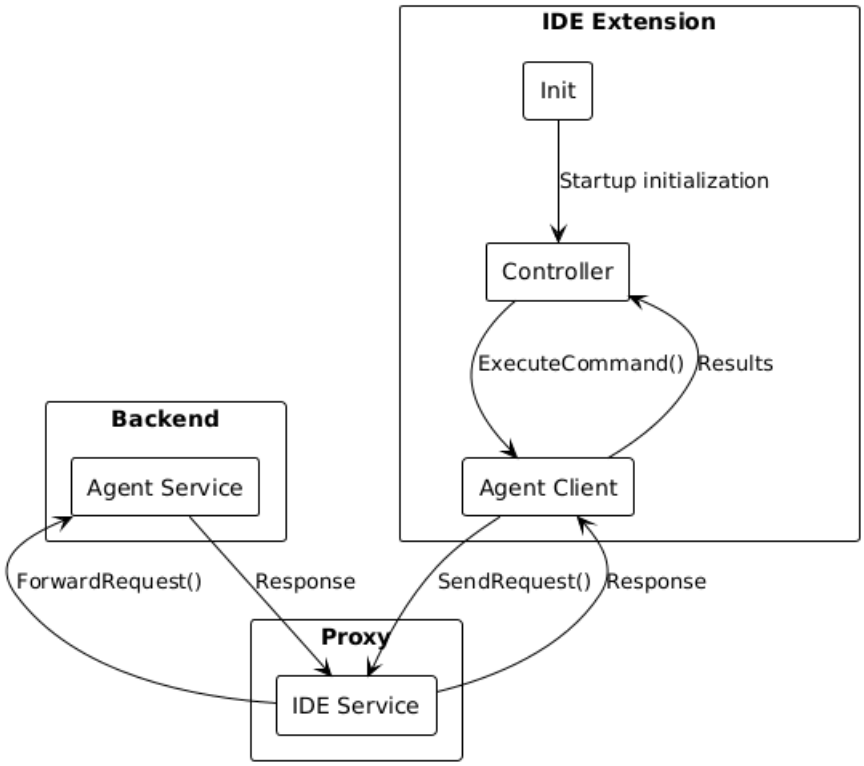

Figure 4.5 – System Architecture: IDE Extension, Proxy, and Backend Communication Flow

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.

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 4.5.

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.

4.3.2.2 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 4.6). This placement ensures high visibility and aligns naturally with the developer's workflow when working on individual files.

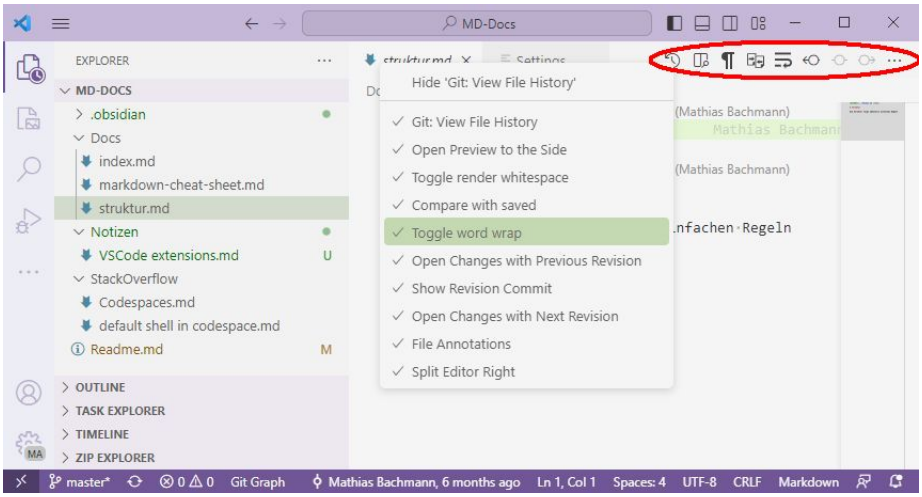

Figure 4.6 – VS Code Interface: Editor Actions (Illustrative).

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 4.7).

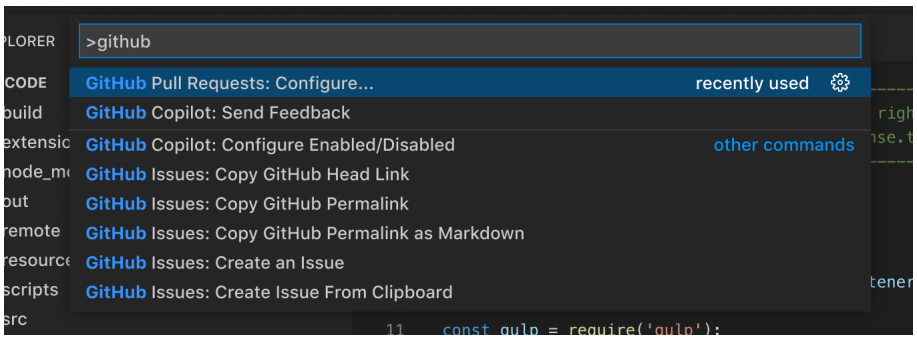

Figure 4.7 – VS Code Command Palette (Illustrative).

Once analysis is triggered, the extension provides immediate feedback via **VS Code notifications** (Figure 4.8). 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.

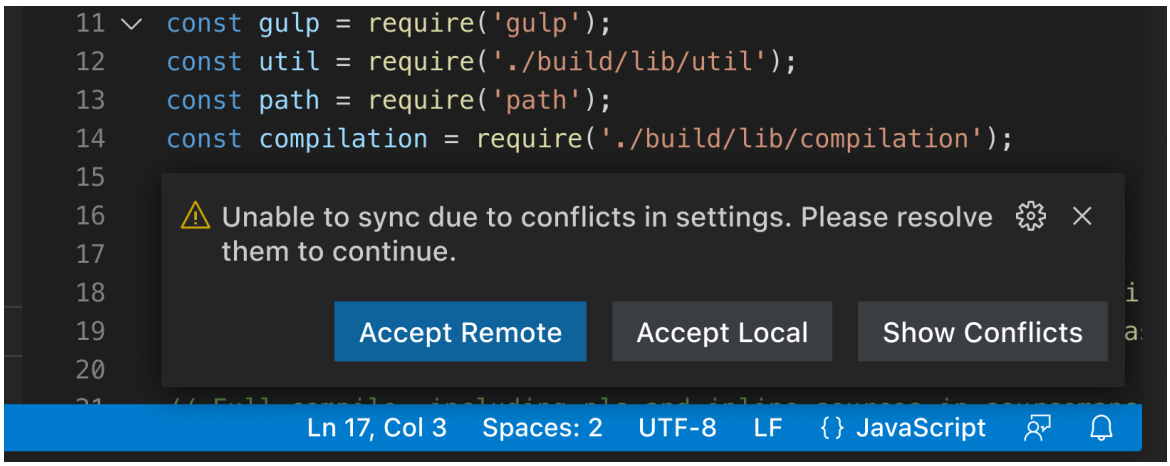

Figure 4.8 – VS Code Notification Interface (Illustrative).

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 4.9). Hovering over the highlighted code reveals the diagnostic explanation, helping developers quickly understand the issue in context.

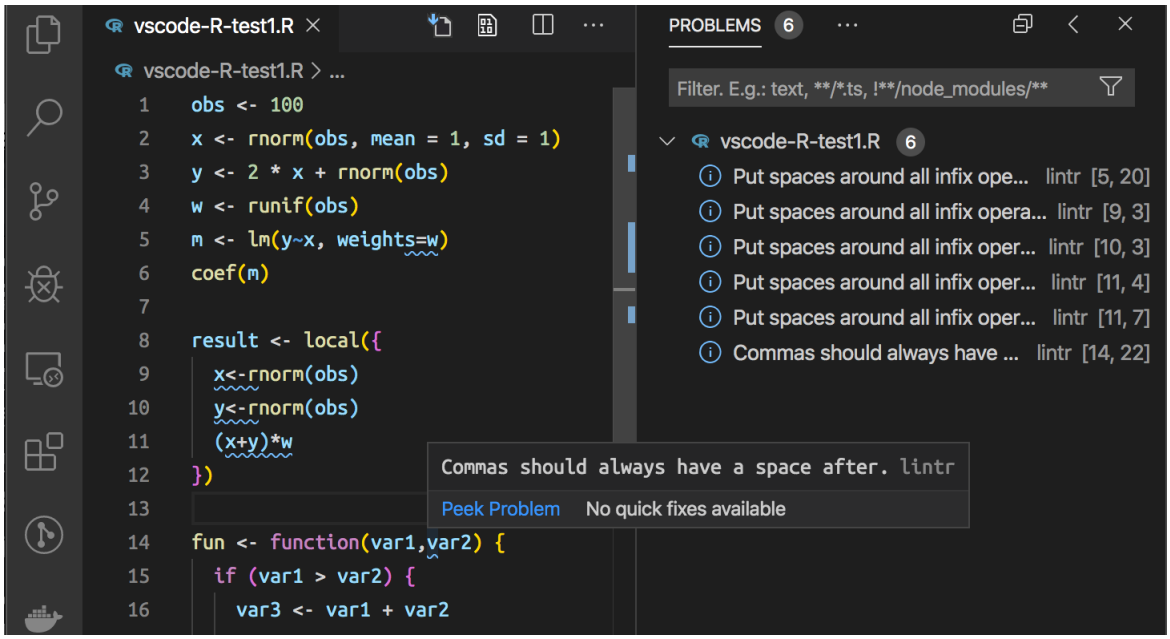

Figure 4.9 – VS Code Diagnostics Interface (Illustrative).

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.

4.3.2.3 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.

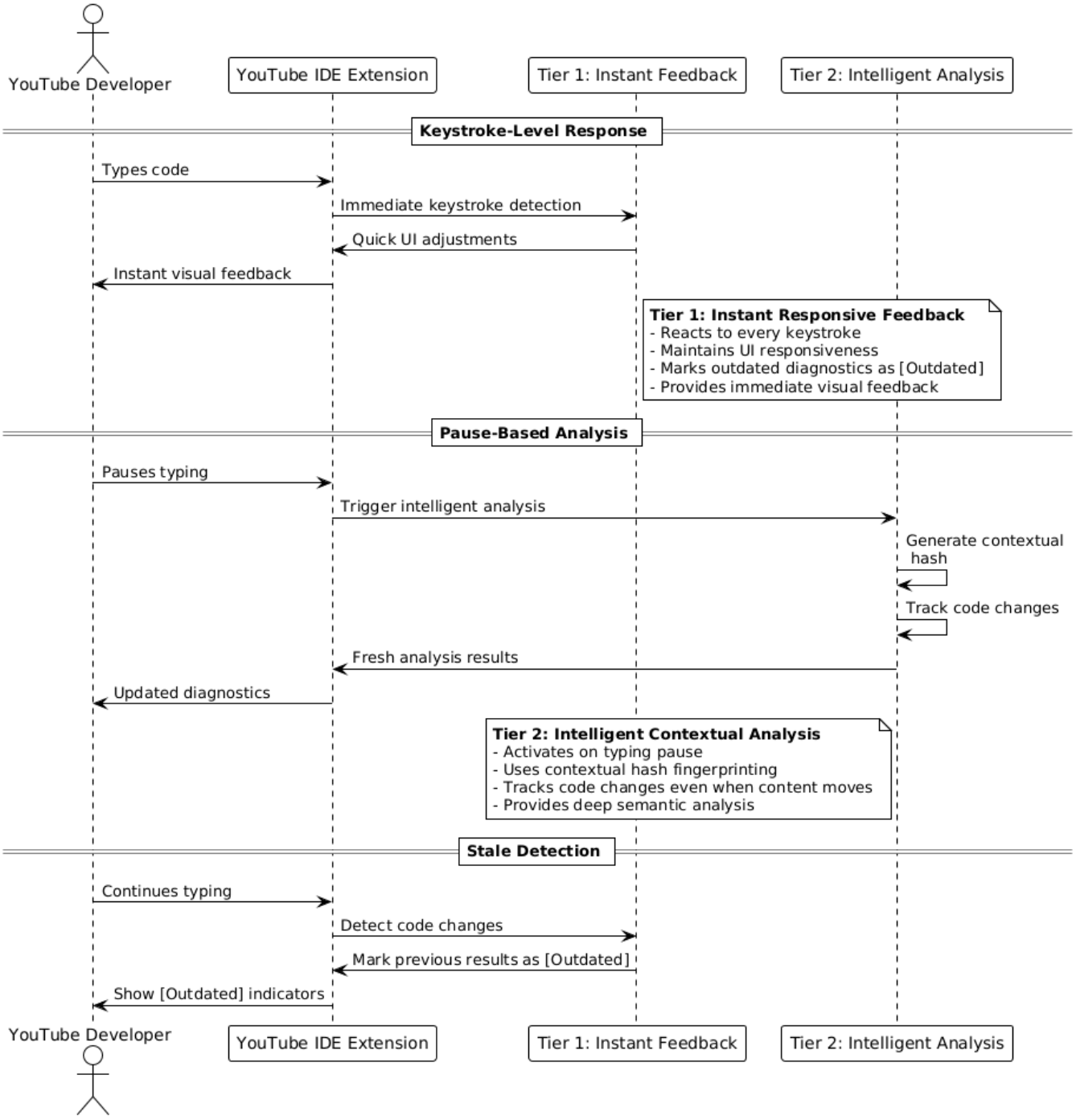

Figure 4.10 – Stale Diagnostics Handling: Two-Tiered System (Illustrative)

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.

4.3.2.4 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 evaluation results, the resulting architecture decision (Parallel Executable with concurrency limiting), and the complete implementation: environment, technologies, and realization across backend and IDE. The system is production-oriented, balancing speed with stability and cost.

Conclusion and Perspectives

This project has developed and implemented an intelligent assistance system integrated into YouTube's internal development environment, aimed at enforcing framework-specific best practices in real time. The system addresses a key challenge in large-scale software engineering: the absence of immediate, contextual feedback on internal frameworks. Unlike existing tools that focus on syntax or public libraries, our solution delivers intelligent, context-aware guidance directly within the developer workflow, reducing technical debt and improving code consistency.

The main contribution consists of an AI agent based on Large Language Models (LLMs) that orchestrates five specialized tools for code analysis, explanation, and fix generation, coupled with a YouTube IDE extension offering intuitive entry points and a two-tier diagnostic state management system that balances responsiveness with precision. Development leveraged Python for the agent, TypeScript and the VS Code API for the extension, and Google's internal AI platform for LLM integration, following an agile Kanban-based workflow.

Design decisions throughout the project were guided by empirical testing and data-driven insights, ensuring that chosen approaches balanced performance, reliability, and usability. Significant technical challenges were addressed, including stale diagnostic handling through a two-tier system, concurrency control for stable performance, and robust semantic evaluation using an LLM-as-a-Judge methodology.

Looking ahead, the next step is implementing a Tiered Analysis Approach that combines a lightweight, rule-based linter for fast detection of simple issues with the LLM agent reserved for complex, subjective cases. This hybrid strategy promises faster, cheaper, and more effective feedback, further enhancing the developer experience. A teammate is already building the linter, and integration plans are in place.

Overall, this project demonstrates the feasibility and value of embedding AI agents into development environments for enforcing framework-specific best practices. It provides a strong foundation for future improvements and represents a meaningful contribution to advancing intelligent developer tools at YouTube.

Appendix : Miscellaneous remarks

فرض أفضل الممارسات مع تكامل نموذج اللغة الكبير وبيئة التطوير المتكاملة

أُنجز هذا المشروع في شركة جوجل زيورخ ضمن متطلبات الدبلوم الوطني للهندسة في هندسة البرمجيات. يدرس دمج الذكاء الاصطناعي التوليدي في تطوير البرمجيات عبر وكيل يعتمد على نموذج اللغة الكبير (LLM) داخل بيئة التطوير المتكاملة (IDE).
يُحلّل الوكيل الشيفرة لاكتشاف انتهاكات معقّدة قد تغفلها أدوات التحليل الثابت، ويُنتج شروحات واضحة واقتراحات عملية تساعد مطوري يوتيوب على رفع جودة الشيفرة والالتزام بالممارسات الفضلى. وباندماجه في سير العمل عن طريق إضافة للبيئة، يعزّز الإنتاجية ويقلّل الديون التقنية دون تعطيل تجربة التطوير.
الكلمات المفتاحية: هندسة برمجيات، ذكاء اصطناعي توليدي، تكامل IDE، جودة الشيفرة، إنتاجية المطورين

Application des meilleures pratiques avec l'intégration LLM-IDE

Ce projet, réalisé chez Google Zurich, intègre un agent basé sur un LLM au sein d'un IDE interne pour soutenir le développement logiciel.

L'agent analyse le code, fournit des explications claires et des suggestions actionnables, et s'intègre au flux de travail par une extension IDE. Il aide les contributeurs YouTube à maintenir une haute qualité de code et à réduire la dette technique sans perturber l'activité.

Mots-clés : Génie logiciel, IA générative, Intégration IDE, Qualité du code, Productivité

Enforcing Best Practices with LLM-IDE Integration

This project, conducted at Google Zurich, embeds an LLM-powered agent inside an internal IDE to support software development.

The agent analyzes code to detect complex violations missed by traditional static tools, producing clear explanations and actionable fixes. Integrated via an IDE extension, it boosts productivity and reduces technical debt without disrupting the developer workflow.

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