



Dhirubhai Ambani University Technology

Formerly DA-IICT

IT314 - Software Engineering

Prof : Saurabh Tiwari



Group 9 : Pull Panda 🐾

(AI-powered Pull Request Reviewer)



Our Team :

Sr No.	Student Name	Student ID
1	PRINCE CHOVATIYA	202301067
2	BHENSADADIA HAPPYBEN CHIMANBHAI	202301077
3	KUSHAL BHUPTANI	202301079
4	HET RANK	202301081
5	AYUSH PATEL	202301084
6	PATEL RITUL JITENDRAKUMAR	202301086
7	PATEL YAKSH PARESHBHAI	202301089
8	PANCHAL YASH KALPESHKUMAR	202301094
9	KANANI KAUSHAL MUKESHBHAI	202301106
10	BHANVADIYA BHAVYA HARESHBHAI	202301123
11	SWAR PATEL	202301132

Github Link : [PULL-PANDA](#)



Table of Contents

Table of Contents	2
Introduction	3
1.1 Purpose	3
1.2 Problem Solved & Benefits	4
1.3 Key Features	5
Stakeholders & Users	6
2.1 Stakeholders	6
2.2 Users	8
Requirements	9
3.1 Functional Requirements	9
3.2 Non-Functional Requirements	12
Elicitation Techniques	14
4.1 Stakeholders and Users	14
4.2 Functional Requirements	15
4.3 Non-Functional Requirements	16
User Stories	17
EPICs	27
6.1 Conflicts Between EPICs and Resolution	34
Sprints	37



1. Introduction

The project is an **AI-powered Pull Request (PR) Agent Reviewer** integrated with GitHub that helps developers and teams review pull requests more efficiently by providing instant, consistent, and automated feedback.

1.1 Purpose

The AI agent acts as a "**first-pass reviewer**" that automatically analyzes pull requests to assist developers by providing instant feedback on key areas such as:

- **Code Quality:** Detects common coding errors, code smells, and style violations based on project standards.
- **Security Issues:** Identifies potential security vulnerabilities and unsafe coding practices.
- **Performance Concerns:** Highlights inefficient code patterns that may affect performance.

- **Project Standards Adherence:** Ensures compliance with project-specific conventions and coding guidelines.

By doing this, the tool enables faster, smarter, and more reliable code reviews, improving development workflow.

1.2 Problem Solved & Benefits

Problems Solved

- **Review Bottleneck:** Manual code review often creates delays when multiple pull requests pile up, especially in large projects or open-source environments.
- **Delayed Feedback:** Waiting for human review can take hours or days, slowing down the development and integration cycles.
- **Inconsistent Reviews:** Different reviewers may apply subjective or inconsistent standards, leading to non-uniform code quality.
- **Missed Issues:** Security vulnerabilities or inefficient coding practices may be overlooked due to human error or time constraints.

Benefits

- **Instant Feedback:** Developers get real-time suggestions and comments as they submit pull requests.

- **Consistency:** Applies uniform rules to all code, reducing variability in reviews.
- **Improved Efficiency:** Automates routine checks, freeing up human reviewers to focus on higher-level architectural decisions.
- **Continuous Learning:** Learns from human reviewer patterns to improve accuracy over time.
- **Customized to Projects:** Adapts to project-specific standards for highly relevant reviews.

1.3 Key Features

- **Automated Code Analysis:** Automatically scans code for errors, security flaws, performance issues, and style violations.
- **Real-Time Integration with GitHub:** Provides feedback directly inside the GitHub pull request interface.
- **Consistent Evaluation Criteria:** Applies the same checks across all pull requests to maintain standard quality.
- **Adaptive Learning:** Improves its suggestions by learning from human review actions and team-specific patterns.

- **Project Customization:** Allows teams to configure rules and coding standards specific to their project needs.
- **Dashboard:** Contains all the information about the user's Repos and PRs. It contains all the information about AI reviews, it's status(merged, pending)



2. Stakeholders & Users

2.1 Stakeholders

1. Software Developers

- Submit pull requests and receive automated feedback.
- Benefit from faster and consistent reviews, allowing them to fix issues early.

2. Code Reviewers / Team Leads

- Perform manual reviews in addition to automated checks.
- Save time by focusing on complex logic and architectural decisions instead of routine issues.

3. Project Managers

- Monitor development progress and ensure code quality standards are met.
- Improve development velocity and reduce bottlenecks in the code review process.

4. Security Engineers

- Ensure that code does not introduce vulnerabilities.
- Benefit from automated detection of common security flaws.

5. Open Source Contributors

- Submit pull requests to open source projects.
- Receive automated guidance on code quality, reducing the need for extensive manual review.

6. DevOps / Integration Engineers

- Ensure smooth integration of code into production pipelines.
- Benefit from more reliable and consistent code being merged.

7. Admin User (System Administrator)

- Responsible for configuring and maintaining the AI-powered Pull Request Agent Reviewer system.

- Manages system settings such as rule configurations, supported programming languages, integration settings
- Ensures the system is running smoothly and updates it as needed

8. End Users

- Benefit from higher software quality, improved performance, and enhanced security of the final product.

2.2 Users

1. Primary Users

- Software Developers
- Code Reviewers / Team Leads

2. Secondary User

- Project Managers
- Security Engineers
- Open Source Contributors
- DevOps / Integration Engineers



3. Requirements

3.1 Functional Requirements

1. Fetch Pull Request (PR)

- The system must automatically fetch pull request details from GitHub whenever a PR is created or updated, using GitHub webhooks and API events.

2. Static Analysis Execution

- The system must run lightweight, multi-language static analysis using tools, ensuring faster execution in CI while maintaining broad rule coverage.

3. RAG-Based Code Review Assistance

- The system must use a Retrieval-Augmented Generation (RAG) pipeline to provide contextual, AI-enhanced review

summaries, suggestions, and explanations based on project-specific documentation and historical reviews.

4. Analyze Code

- The system must analyze code changes in the pull request for:
 - Code quality issues (syntax errors, code smells, style violations).
 - Security vulnerabilities (e.g., insecure functions, hard-coded secrets).
 - Performance concerns (e.g., inefficient loops, excessive memory usage).
 - Adherence to project-specific coding standards.

5. Generate Review Report

- The system must generate a detailed review report summarizing detected issues, security risks, performance suggestions, and coding standard violations.

6. Post Comments on PR

- The system must post automated review comments directly on the GitHub pull request, highlighting issues inline with the code.

7. Support Multiple Programming Languages

- The system must support at least commonly used languages such as Python, JavaScript, and Java, and allow easy addition of new language support.

8. Dashboard for Monitoring

- The system must provide a dashboard where users can:
 - View PR analysis summaries
 - Review historical insights
 - Inspect issue trends
 - Filter reports by repository, PR, or author

9. Follow GitHub Workflow

- The system must seamlessly integrate with GitHub's pull request workflow using webhooks, ensuring that analysis is triggered automatically on PR creation and updates.

10. Custom Reviews and Rules Configuration

- The system must allow teams to configure custom coding standards, custom rules, and thresholds that match their project-specific requirements.

11. Track Metrics of Performance

- The system must log and track performance metrics, including time taken for analysis, number of issues found, and trends over time.

12. Support Multiple Repository Types

- The system must work with both public and private repositories and support various project types.

13. Integrate with CI/CD Pipelines

- The system must support integration with Continuous Integration / Continuous Deployment pipelines to automatically enforce code quality gates before merging.

14. Reviewer Controls

- The dashboard must offer admin-level controls for:
 - Reviewing logs
 - Managing rule sets
 - Monitoring PR activity

- Evaluating system performance and accuracy

3.2 Non-Functional Requirements

1. Performance & Scalability

The system should analyze pull requests and return feedback in a reasonable time window suitable for active development workflows. It must scale smoothly to handle multiple concurrent PRs across repositories without noticeable performance drop.

2. Accuracy & Reliability

The system should aim to minimize false positives and false negatives while adapting over time to reviewer patterns. It must remain consistently available, handle failures gracefully, and recover without affecting ongoing operations.

3. Usability & Compatibility

The system should deliver clear, actionable feedback directly within the GitHub pull request interface. It must support both public and private repositories and integrate seamlessly with commonly used CI/CD platforms.

4. Auditability

All analysis activities, detected issues, and rule evaluations should be securely logged. Logs must remain accessible to authorized administrators for compliance, monitoring, and debugging.

5. Maintainability

The system should follow a modular structure to allow easy updates, such as adding new languages, custom rules, or external integrations. Enhancements should be achievable without disruptive changes to the overall architecture.

6. Security

The system must ensure secure handling of authentication and repository access. All communications should be encrypted, and sensitive credentials must be strictly protected with controlled access.



4. Elicitation Techniques

4.1 Stakeholders and Users

- **Elicitation Techniques Used:**
 - **Identified Stakeholders**

Identified the primary and secondary stakeholders by studying the project scope (e.g., Admin, Developer, Project Manager).
 - **Questionnaires/Surveys**

Collected feedback from potential users (developers, admins, reviewers) through structured questionnaires to understand their expectations from an automated pull request reviewer.
 - **Use Cases and Scenarios**

Developed example scenarios of pull request submission, review, and reporting to clearly understand who interacts with the system and how.

4.2 Functional Requirements

- **Elicitation Techniques Used:**

- **Stakeholder Analysis**

Identified the primary and secondary stakeholders by studying the project scope (e.g., Admin, Developer, Project Manager).

- **Questionnaires/Surveys**

Asked developers and admins through structured questionnaires what actions they expect the system to perform (e.g., submit PR, configure rules, get instant feedback).

- **Task Observation**

Observed existing manual PR review workflows to identify pain points like delayed feedback or inconsistent reviews.

- **Use Cases and Scenarios**

Defined step-by-step workflows (e.g., submit PR → analyze → comment → merge) to capture all necessary system behaviors clearly.

4.3 Non-Functional Requirements

- **Elicitation Techniques Used:**

- **Brainstorming**

Gather ideas about important qualities the system should have (performance, security, usability).

- **Questionnaires/Surveys**

Asked stakeholders about system expectations (e.g., fast response, reliability, easy-to-use interface).

- **Background Reading**

Studied best practices for automated code review systems, CI/CD integration, and security measures to understand typical non-functional constraints.



5. User Stories

User Story 1

Developer Submits Pull Request

Front of Card:

As a developer,
I want to submit a pull request to GitHub,
So that the AI can automatically analyze my code changes.

Back of Card:

1. The system fetches the PR and begins automatic analysis.
2. If the PR cannot be fetched (e.g., network issue, incorrect repo permissions), the system logs the error and notifies the admin.

Acceptance Criteria

- The system successfully fetches the PR after submission or update.
- If fetching fails, an error is logged and the admin is notified.
- The system triggers analysis automatically without manual intervention

User Story 2

Receive Feedback

Front of Card:

As a developer,

I want to receive automated review comments directly on my pull request,

So that I can quickly fix code quality, security, or performance issues before merging.

Back of Card:

1. If any analysis tool fails, is not triggered, or encounters unsupported syntax, the system posts a direct comment on the PR explaining the failure.

Acceptance Criteria

- Automated comments appear on the GitHub pull request shortly after submission or update.
- Comments highlight issues related to code quality, security, performance, and project standards.

User Story 3

Configure Custom Rules

Front of Card:

As an admin,

I want to configure custom project-specific coding standards and security rules,

So that the automated reviews reflect the unique requirements of our project.

Back of Card:

1. Admin can set and save rules that affect future PR reviews.

Acceptance Criteria

- Admin can add, edit, and delete custom rules via a configuration interface.
- Invalid configurations are rejected.
- New rules are applied to all future PR analyses without system downtime.

User Story 4

Focus on Complex Logic by Human Reviewers

Front of Card:

As a code reviewer,

I want to focus only on high-level architectural or business logic decisions,

So that the AI handles routine checks and saves me time.

Back of Card:

1. AI filters out style and security issues, and presents a clean PR ready for human review.
2. If the AI misses a basic error, the reviewer manually flags it during their review.

Acceptance Criteria

- The AI automatically filters out simple style and security issues.
- Human reviewers see only non-trivial architectural or business logic issues.
- The system logs any issues missed by the AI that are later caught by reviewers.

User Story 5

Track Performance Metrics

Front of Card:

As a project manager,

I want to track key metrics such as average review time and number of issues found,

So that I can monitor code quality trends and improve development processes.

Back of Card:

1. Metrics are available in dashboards, which display trends and summaries.

Acceptance Criteria

- A dashboard shows key metrics like average review time, number of issues per PR, and trend graphs.
- Metrics update in real-time or near real-time after each analysis.

User Story 6

Multi-Language Support

Front of Card:

As a developer,

I want the system to support multiple languages like Python, JavaScript, and Java,

So that I can get automated reviews regardless of the language I use.

Back of Card:

1. The system correctly analyzes supported languages and provides relevant feedback.
2. If a language is unsupported, the system informs the developer that automated analysis is not available for that PR.

Acceptance Criteria

- The system successfully analyzes supported languages (Python, JavaScript, Java).
- If an unsupported language is detected, a clear message appears in the PR indicating it is not analyzed.

User Story 7

CI/CD Pipeline Integration

Front of Card:

As a DevOps engineer,

I want to integrate the automated reviewer into CI/CD pipelines,
So that reviews run automatically during builds to prevent merging
broken code.

Back of Card:

1. The system runs reviews as part of the CI/CD workflow and prevents merge if critical issues are found.
2. If the integration fails, the CI pipeline fails gracefully and logs the issue for admin intervention.

Acceptance Criteria

- The system successfully analyzes supported languages (Python, JavaScript, Java).
- If an unsupported language is detected, a clear message appears in the PR indicating it is not analyzed.
- Adding a new language configuration does not require major system downtime.

User Story 8

Audit Logs for Compliance

Front of Card:

As an admin,

I want the system's analysis operations to be visible through GitHub's built-in logs,

So that I can audit activity directly from the PR without maintaining a separate logging dashboard.

Back of Card:

1. All analysis steps, tool outputs, and decisions are recorded automatically in GitHub's PR timeline and workflow logs.
2. If logging or workflow execution fails, GitHub marks the run as failed and notifies the admin or repository maintainers.

Acceptance Criteria

- GitHub PR timeline shows events such as analysis triggered, comments posted, and status checks.
- GitHub Actions workflow logs show Semgrep results, failures, and processing steps.
- Admin does **not** need a separate dashboard – all logs are available directly on GitHub.
- If a workflow or log generation fails, GitHub automatically marks the run as failed and notifies maintainers.

User Story 9

Secure System Access and Data Handling

Front of Card:

As an admin,

I want the system to securely manage authentication and sensitive data,
So that the system is protected against unauthorized access and data
breaches.

Back of Card:

1. API tokens and credentials are securely stored and encrypted.
2. Any unauthorized access attempt is blocked and logged for investigation.

Acceptance Criteria

- All API communications use HTTPS.
- API tokens and sensitive data are stored encrypted.
- Unauthorized access attempts are blocked and logged.

User Story 10

AI-Assisted Code Summaries

Front of Card:

As a developer,

I want concise AI-generated summaries for each PR,

So that I can quickly understand major changes.

Back of Card:

1. The RAG system summarizes code modifications, dependencies, and reasoning.

Acceptance Criteria

- PR summary appears in dashboard or PR comment.
- Summary highlights important changes, risks, and patterns.
- Failures fall back to a minimal description.

User Story 11

Dashboard Insights

Front of Card:

As a developer or manager,

I want a dashboard showing RAG-based insights and aggregated findings,

So that I can understand project-wide issue trends.

Back of Card:

1. Dashboard shows combined Semgrep + RAG insights, issue clusters, and patterns.

Acceptance Criteria

- Dashboard displays trends, severity distribution, and issue categories.
- Filters available by repo, PR, author, and issue type.
- Updates occur automatically after each analysis.

User Story 12

Configurable RAG Knowledge Base

Front of Card:

As an admin,

I want to manage the knowledge base used for RAG,

So that the AI reflects our latest guidelines, docs, and patterns.

Back of Card:

1. Admin can upload new documentation or delete outdated references.

Acceptance Criteria

- Admin can update RAG documents.
- Changes reflect in the next analysis cycle.
- Invalid document formats are rejected.



6. EPICs

Epic 1:

Pull Request Analysis

- **Goal:**

Automatically analyze submitted pull requests to ensure code quality, security, performance, and provide intelligent summaries.

- **Description:**

The system fetches pull requests from GitHub, performs automated analysis (code quality, security, performance, standards), generates inline review comments, and provides AI-generated PR summaries. Human reviewers get only high-level issues, while routine checks are auto-handled.

- **User Stories Covered:**

#1 Developer Submits Pull Request

#2 Receive Instant Feedback

#4 Focus on Complex Logic by Human Reviewers

#10 AI-Assisted Code Summaries

Epic 2:

Custom Rule Management

- **Goal:**

Enable admins to configure and manage project-specific analysis rules and RAG knowledge.

- **Description:**

Admins can create, edit, and delete custom coding standards, security rules, performance thresholds, and maintain the RAG knowledge base. All configurations are validated before being applied, and updates affect future analyses.

- **User Stories Covered:**

#3 Configure Custom Rules

#12 Configurable RAG Knowledge Base

Epic 3:

Metrics and Reporting

- **Goal:**
Track review performance metrics and show dashboard insights.
- **Description:**
The system provides dashboards showing key metrics such as review time, issues per PR, trends, aggregated insights, RAG analysis patterns, and allows exporting. Metrics update after each analysis, and logging failures raise alerts.
- **User Stories Covered:**
#5 Track Performance Metrics
#11 Dashboard Insights

Epic 4:

Multi-Language Support

- **Goal:**

Support code review in multiple programming languages.

- **Description:**

The system can analyze pull requests in different supported languages (e.g., Python, JavaScript, Java). If a language is unsupported, the system clearly notifies the developer that automated analysis is not available.

- **User Stories Covered:**

#6 Multi-Language Support

Epic 5:

CI/CD Integration

- **Goal:**

Integrate the automated review system into CI/CD pipelines to ensure code quality is enforced before merging.

- **Description:**

The system integrates with tools like GitHub Actions to run automated reviews during build steps, blocking merges when critical issues are detected and allowing warnings for transient errors.

- **User Stories Covered:**

#7 Integrate with CI/CD Pipeline

Epic 6:

User Access and Security

- **Goal:**

Manage secure system access and protect sensitive data.

- **Description:**

The system enforces secure authentication, encrypted storage of credentials, HTTPS communication, and detection of unauthorized access attempts. Role-based access ensures only authorized users can configure rules or modify system settings.

- **User Stories Covered:**

#9 Secure System Access and Data Handling

Epic 7:

Audit and Compliance

- **Goal:**
Ensure complete traceability and compliance by maintaining structured, immutable logs.
- **Description:**
All analysis actions, configuration changes, and notifications are logged in an immutable manner with timestamps, and accessible to admins for audit purposes.
- **User Stories Covered:**
#8 Audit Logs for Compliance

6.1 Conflicts Between EPICs and Resolution

Conflict 1:

Pull Request Analysis ↔ Custom Rule Management

- **Reason:**

Pull Request Analysis uses built-in rules to automatically check code, but Custom Rule Management lets admins add their own custom rules.

- **Conflict:**

Custom rules could accidentally conflict with the built-in rules, causing inconsistent results in the automated review.

- **Resolution:**

Make sure the system gives priority to core rules, and checks custom rules for errors before applying them.

Conflict 2:

CI/CD Integration ↔ User Access and Security

- **Reason:**

CI/CD Integration needs the system to run automatically during builds, while User Access and Security control who can access and configure the system.

- **Conflict:**

Strict access rules might prevent the CI/CD pipeline from running automated reviews.

- **Resolution:**

Create special service accounts with just the right permissions for CI/CD, separate from human user accounts.

Conflict 3:

Metrics and Reporting ↔ Audit and Compliance

- **Reason:**

Metrics show easy-to-read summaries of system performance, while Audit keeps detailed records of every action.

- **Conflict:**

Generating metrics separately could skip the audit log, making it hard to trace what happened.

- **Resolution:**

Always calculate metrics based on the audit logs to keep everything traceable and reliable.



7. Sprints

Sprint 1 – Cloud LLM MVP (Version 1)

1. GitHub PR Fetch

- The system connects to the GitHub API to automatically retrieve pull request details, including files changed, additions, deletions, and overall metadata.
- This ensures that the model receives accurate, real-time information directly from the repository.

2. Send Diff + Metadata to Cloud LLM

- Once the PR diff and metadata are collected, they are packaged into a structured input prompt.
- This payload is then sent to a cloud-hosted LLM for analysis, ensuring scalable and high-quality review generation.

3. Return Structured PR Review

- The cloud LLM responds with a formatted, structured review containing suggested improvements, issues detected, and general feedback.

- This structured format ensures consistency and helps developers quickly understand the model's recommendations.

4. Basic Pipeline + Simple UI Integration

- The end-to-end pipeline connects GitHub, the backend processing layer, and the cloud LLM into a working MVP flow.
 - A minimal UI or GitHub comment interface displays the LLM-generated review back to the user, completing the workflow.
-

Sprint 2 – Static Analyzers (Version 1.1, 1.2, 2)

1. GitHub Actions Automation (.yml)

- Trigger review on pull_request events
- Backend generates review → auto-comments on PR

2. Static Analyzer Integration

- Used Semgrep static analysis
- Inject analyzer outputs into prompts

3. Iterative Prompt Selector (Random Forest Regressor v1)

- First ML-based prompt selector

- Trained on synthetic PR datasets
 - Predicts the best-performing prompt template
-

Sprint 3 – RAG with Repo Indexing (Traditional Rag)

- Index full repository
 - Generate embeddings using cloud model
 - Retrieve context-specific files during PR review
 - Improve comment accuracy with surrounding code context
 - Local (dev) vector database used temporarily (e.g., Chroma local, FAISS)
-

Sprint 4 – Advanced Optimization + Fine-Tuning (Updated) (version 1.3, RAG_version1.3)

This sprint now includes four major upgrades:

Upgrade Iterative Prompt Selector → SGD (Online Learning)

- Replace Random Forest with SGDRegressor
- Supports `partial_fit`
- Continuously improves as more PRs flow in
- Allows streaming, real-time prompt optimization

RAG Vector DB Migration (Local → Online)

In Sprint 3 we used a local DB (e.g., local Chroma, FAISS). Now upgraded to an online, production-grade vector DB using Pinecone Benefits:

- Fast retrieval across repos
- Scalable embeddings storage
- No local storage bottlenecks
- Reliable multi-user PR context indexing

End-to-End Optimization

- RAG + fine-tuned model integration
 - Prompt selector + AI review engine unified pipeline
 - Automated performance evaluation on test PRs
-

Final Sprint – Dashboard + Documentation Testing (RAG_Version1.3)

Includes all the final polish:

Dashboard Enhancements

- PR history

Documentation

- Finalizing overall documentation of project and summary reports.

Testing Suite

- Static Testing
- Unit Testing (with regression)
- Integration Testing
- Black Box Testing
- Non-functional Testing
 - Load Testing
 - Spike Testing
- GUI testing

