AI migration platform: An Ai-Powered Platform for Automated Api Migration and Validation

Ms. P. Drishya[1], Nithish Krishna S[2], PranishN[3], Ronald R[4]
[1,2,3,4]*Department of CSE (IoT & Cybersecurity Including BCT),*
SNS College of Technology, Coimbatore, India

***Abstract*—Third-party application programming interfaces (APIs) evolve as providers change methods, parameters, response structures and authentication requirements. These changes can leave existing applications incompatible and require developers to inspect release notes, locate affected calls, update source code and validate behaviour manually. This paper presents AI Migration Platform, an AI-assisted platform intended to organise these tasks into a reviewable workflow. The project report describes a web application that connects to GitHub repositories, scans supported source files for candidate outdated API usage, generates candidate transformations with a large language model, validates proposed changes where checks are configured, and prepares a diff or pull request for developer review. The identified technology stack includes React, TypeScript, Vite and Tailwind CSS on the frontend; Node.js, Express and TypeScript on the backend; SQLite with Prisma for persistence; Google Gemini for AI-assisted transformation; and GitHub REST API integration through Octokit. The report's presentation lists 12 repositories connected, 78 outdated API instances, 84 files migrated and 128 validations. These are presentation-reported demonstration metrics, not independently benchmarked production results. The paper describes the need, related concepts, architecture, methodology, reported results and limitations of the prototype.**

***Index Terms*—API migration, automated refactoring, large language models, static analysis, software maintenance, GitHub, validation.**

I. INTRODUCTION

A. Background

Modern software applications use third-party APIs and software development kits (SDKs) for services such as artificial intelligence, payments, communications, identity and analytics. The provider controls the API contract; therefore, applications must adapt when methods are removed, parameters change, response formats are modified or authentication requirements evolve. A package version update alone does not guarantee that source-code call sites remain compatible.

A migration may involve more than renaming a method. It can require changes to input arguments, return-value handling, error management and application logic. Developers must locate all affected uses, interpret provider documentation, update code and confirm that existing behaviour remains correct. Across multiple repositories, this work can become repetitive and error-prone.

The AI Migration Platform uses a combined workflow in which repository scanning identifies candidate call sites, AI assistance proposes context-aware edits, validation results are collected, and a developer reviews the diff before accepting the migration. The platform is intended to reduce repetitive discovery and editing effort without removing human responsibility for correctness.

B. Contributions and Organisation

The project contribution is an integrated prototype concept that connects repository access, source scanning, AI-assisted transformation, validation and review preparation. Section II explains the necessity of the project; Section III summarises related concepts; Section IV describes the architecture; Section V presents the methodology; Section VI discusses implementation; Section VII reports demonstration metrics and evaluation considerations; Section VIII outlines limitations; and Section IX concludes with future work.

II. NECESSITY OF THE PROJECT

A. Cost of Manual API Migration

In a typical maintenance workflow, developers monitor provider announcements, search source files for obsolete calls, read migration guides, edit the affected code, run tests and submit changes for review. These steps consume engineering time and can be inconsistently performed when many repositories or services are involved. Missed call sites can remain undetected until an application fails at build time or runtime.

B. Gaps in Existing Approaches

Dependency-update tools help teams track package releases and can propose version changes, but they may not rewrite source code that calls removed methods. Text search can identify matching strings but may not understand code structure or the intended replacement. Static analysis can detect syntax and structural patterns, but semantic migrations may require context about the provider contract and application behaviour. AI-only rewriting is flexible but may produce plausible yet incorrect code.

C. Problem Statement

The problem addressed is the absence of a single assisted workflow that helps developers detect potentially outdated API usage across connected repositories, propose source-code migrations, validate the generated changes and organise the results for human review. The project aims to reduce repetitive migration effort while keeping correctness checks and developer approval central.

D. Objectives

- Connect or select a GitHub repository as the source of project code.
- Scan supported source files to identify candidate outdated API usage.
- Generate candidate transformations with AI assistance while attempting to preserve surrounding business logic.
- Run available syntax, structural, dependency and security checks.
- Present explanations, diffs and validation information for review.
- Prepare a structured change set or pull request where the integration and permissions permit.
- Record scan and migration-job information for traceability.

Need	Gap in current practice	Project response
Detection	Manual search across files	Repository scanning for candidate usages
Migration	Developers interpret guides and edit code	AI-assisted candidate transformation
Quality	Checks are scattered across tools	Validation pipeline and review summary
Safety	Generated edits may be accepted without context	Diff-based human review before acceptance

III. RELATED WORK AND TECHNICAL CONTEXT

A. API Evolution and Migration Guides

API providers publish versioned documentation, deprecation notices and migration guides describing changes to endpoints, methods, parameters, responses and authentication. These resources provide the primary evidence for intended replacements. Applying the guidance to every call site still requires repository-specific analysis.

B. Dependency Update Automation

Tools such as automated dependency update services can identify newer package versions and prepare reviewable update pull requests. However, changing a dependency version does not necessarily transform application code that relies on an older interface. AI Migration Platform extends the general update workflow toward source-code usages associated with API changes.

C. Static Analysis and Abstract Syntax Trees

Static analysis examines source code without executing it. Abstract syntax trees (ASTs) represent code as structured nodes and can distinguish a function call from similar text in a comment. These techniques can help locate candidate usages and detect malformed code, but they cannot independently prove that the transformed program preserves intended business behaviour.

D. AI-Assisted Code Transformation

Large language models can use surrounding source code and migration instructions to propose context-sensitive edits, including changes to arguments, response handling or error management. Their output remains probabilistic and may contain unsupported assumptions. Generated changes should therefore be treated as candidates and checked against provider documentation and project tests.

E. Research Gap

The project's motivation is to connect detection, migration generation, validation and review preparation in one workflow. Its hybrid direction combines static analysis for locating candidates, AI assistance for proposing transformations, deterministic checks where available and human review for acceptance. The report's literature survey presents these as relevant technical concepts; it does not claim that every mentioned tool is integrated into the prototype.

IV. SYSTEM DESIGN

A. High-Level Architecture

The frontend provides repository selection, scan status, migration results and review interfaces. Backend services coordinate repository access, source scanning, AI requests, validation and persistence. GitHub provides repository operations and pull-request workflows; Gemini supports AI-assisted transformation; and SQLite with Prisma is identified as the persistence layer.

Developer → React / TypeScript Frontend → Express / TypeScript Backend → Repository Scanner → AI Migration → Validation → Reviewable Diff / Pull Request

The database model described in the project report includes conceptual entities such as User, Repository, RepositoryScan, MigrationRule, MigrationJob and ApiProvider. The exact implemented schema should be checked against the current Prisma schema before final publication.

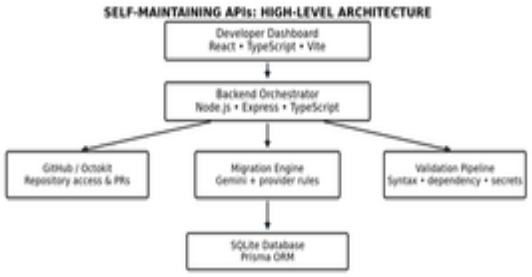

Fig. 1. High-level architecture of the AI Migration Platform.

B. Technology Stack

Layer	Technology identified	Purpose
Frontend	React, TypeScript, Vite, Tailwind CSS	Interactive user interface
Backend	Node.js, Express, TypeScript	REST endpoints and workflow coordination
Persistence	SQLite, Prisma ORM	Structured application records
AI integration	Google Gemini API	Candidate migration generation and explanation
GitHub	GitHub REST API, Octokit	Repository access and review workflow
Automation	n8n, optional / where configured	Follow-up workflow automation

C. Functional Requirements

The project report identifies repository selection, scanning, AI transformation, diff presentation, validation reporting, job-status recording, pull-request preparation and developer approval as core requirements. Reliability, security, maintainability, traceability and usability are non-functional requirements. Credentials should be kept outside source control, logs should avoid secrets, and repository permissions should be limited to required operations.

V. METHODOLOGY

A. End-to-End Workflow

1. Connect or select a repository and obtain permitted source access.
2. Scan supported files and identify candidate outdated API usages with location context.

3. Collect relevant source code and migration guidance for each finding.
4. Ask the AI migration component to propose a candidate transformation.
5. Run the validation checks configured for the project and record their outcomes.
6. Present the original and proposed code, explanation, warnings and validation summary.
7. Require a developer to accept, revise or reject the candidate migration.
8. Prepare a pull request where GitHub integration and permissions allow.

B. Hybrid Migration Strategy

The report compares four approaches. Manual migration preserves developer control but is slow and repetitive. Rule-based rewriting is predictable for known patterns but limited to supported rules. AI-only transformation can use context but carries hallucination and regression risk. The selected approach is a hybrid pipeline that combines scanning, AI-assisted generation, available checks and human review.

C. Validation Pipeline

Validation should be layered. Syntax checks identify malformed output; AST or structural checks compare relevant code structure where supported; dependency checks inspect package consistency; security scans flag configured secret patterns; and tests assess behaviour when a test environment is available. A failed check or low-confidence result should be surfaced rather than silently treated as safe. The report lists these checks as intended/configured validation categories, so claims of execution must be supported by current logs.

Migration outcome = candidate transformation + validation evidence + developer decision

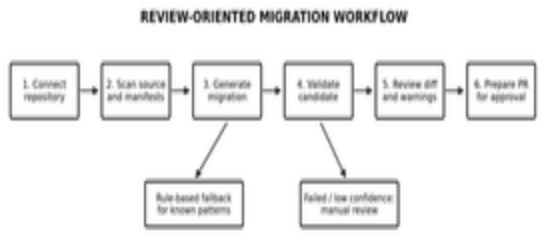

Fig. 2. Review-oriented API migration workflow.

VI. IMPLEMENTATION

A. Frontend and Backend

The project report identifies a web-based frontend built with React, TypeScript, Vite and Tailwind CSS. Backend endpoints and workflow coordination are associated with Node.js, Express and TypeScript. The application is intended to expose repository connection, scanning progress, candidate migrations, changed-file diffs and validation outcomes through a single interface.

B. Repository Scanning and Migration Generation

The scanner searches supported source files for candidate outdated API usage. The migration stage uses source context and migration guidance to generate proposed edits. The presentation identifies JavaScript, TypeScript and Python as target languages; the precise parser and end-to-end support for each language should be verified against the current build. The report also describes provider-specific migration rules as a potential way to make known transformations more repeatable.

C. GitHub and Persistence

GitHub REST API integration through Octokit is identified for repository operations and pull-request preparation. SQLite with Prisma is identified for storing application records, including repositories, scans and migration jobs. A live pull request depends on valid credentials, suitable repository permissions and a working integration. If the demonstration uses simulated results, they must be labelled as simulated rather than live GitHub operations.

D. Security and Maintenance

The report discusses credential handling, least-privilege access, avoiding secrets in logs, session controls and encryption as security considerations. These controls should only be described as implemented after checking the running source code. Ongoing maintenance includes updating provider rules, keeping parsers and dependencies current, adapting to external API changes, improving prompts, and reviewing false positives and missed detections.

VII. EVALUATION AND RESULTS

A. Demonstration Overview

The project report describes a demonstration workflow involving repository connection, detection of outdated API usage, AI-assisted migration, validation and pull-request preparation. It also cautions that the exact status of each end-to-end capability should be verified against the latest running build and saved logs.

B. Presentation-Reported Metrics

Metric	Reported value	Interpretation in report
Repositories connected	12	Repositories in the reported demo cluster
Outdated API instances	78	Candidate instances reported by scanning
Files migrated	84	Files reported as transformed
Validations	128	Validation runs/checks reported; pass meaning needs logs

These figures are copied from the project presentation and are not independently rerun or verified while preparing this paper. In particular, “128 validations” must not be interpreted as 128 successful checks unless the underlying results establish that. Counts of files or findings alone do not prove migration correctness.

C. Evaluation Metrics Needed

A rigorous evaluation should use repositories with known breaking API changes and independently reviewed expected migrations. Useful metrics include detection precision and recall, proportion of generated migrations accepted by reviewers, syntax-validation pass rate, test pass rate, false-positive rate, regression rate and end-to-end processing time. The project report does not provide a reproducible labelled benchmark or sufficient logs to calculate these metrics here.

D. Evidence to Attach

For a final submission, include genuine screenshots or logs from the current build: repository connection, scan findings, before/after diff, validation output and a generated pull request if live creation was tested. Each figure should have a numbered caption and a short explanation. Placeholder screenshots should not be presented as experimental evidence.

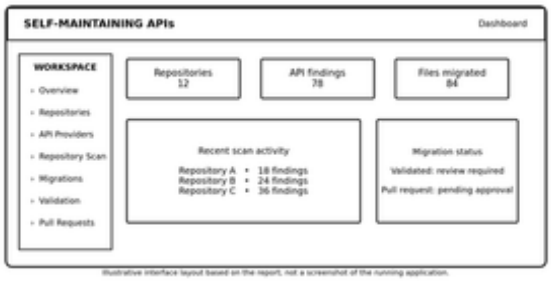

Fig. 3. Illustrative dashboard layout based on the project report; not a live screenshot.

VIII. DISCUSSION AND LIMITATIONS

A. Strengths

- Combines repository analysis, candidate migration, validation and review preparation in one workflow.
- Uses AI assistance for context-sensitive transformations rather than relying only on text replacement.
- Keeps developer review as a safety gate before accepting changes.
- Provides a structure for recording scan and migration-job information.
- Uses a technology stack based on common web, database, AI and GitHub integration components.

B. Limitations

- The report does not establish benchmark-level accuracy or production reliability.
- AI-generated changes can be incorrect or alter behaviour in ways static checks miss.
- Language support depends on the parser and migration paths actually configured.
- Provider documentation and migration rules may be incomplete or become outdated.
- Repositories may lack tests covering affected integrations.
- GitHub pull-request automation depends on working integration, credentials and permissions.
- Security, encryption, validation and automation features must be verified in the current source code before being claimed as implemented.

C. Safety and Human Review

A syntactically valid change may still be semantically wrong. The platform should present the diff, rationale, warnings and test results, and require explicit developer approval. Low-confidence changes and failed checks should be escalated for review. Automated pull-request preparation should not be described as automatic safe merging.

Current capability area	Limitation / evidence requirement
Repository scanning	Confirm supported file types and current scan logs
AI migration	Review correctness against provider documentation
Validation	Record which checks actually ran and whether they passed
Pull request	Confirm live GitHub creation rather than simulated output
Reported metrics	Retain source logs and define each metric precisely

IX. CONCLUSION AND FUTURE WORK

The AI Migration Platform addresses a recurring software maintenance challenge: third-party API changes can make existing source-code integrations outdated or incompatible. The project report describes an AI-assisted workflow that connects repository scanning, candidate migration generation, validation and review preparation. Its identified stack combines React, TypeScript, Vite, Tailwind CSS, Node.js, Express, SQLite, Prisma, Gemini and GitHub integration through Octokit.

The report's presentation lists 12 connected repositories, 78 outdated API instances, 84 migrated files and 128 validations. These remain presentation-reported demonstration figures rather than independently verified research results. Further evaluation is required to establish migration accuracy, regression risk and performance across varied repositories.

Future work includes expanding provider and language coverage, building a curated migration-rule library, adding unit-test generation and stronger CI integration, linking changes to exact provider documentation and source call sites, improving rollback and audit history, strengthening security controls, and evaluating the platform on a labelled benchmark with precision, recall, validation pass rate and manual correction effort.

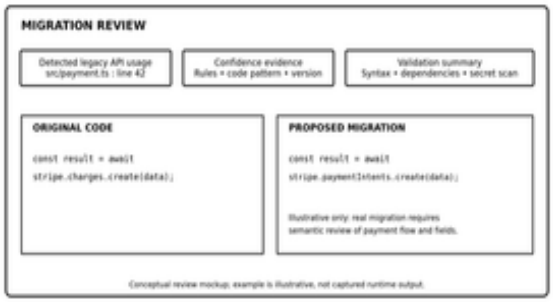

Fig. 4. Illustrative migration-review view; code example is schematic.

APPENDIX: MIGRATION WORKFLOW

The intended workflow can be summarised as follows: (1) obtain permitted repository context; (2) scan supported files and record candidate API call sites; (3) associate findings with provider guidance and migration rules; (4) generate a proposed edit; (5) run configured syntax, structural, dependency and security checks; (6) prepare a diff with warnings and validation evidence; and (7) require developer acceptance before the migration is treated as complete. This sequence is a high-level description of the project workflow, not a substitute for execution logs or proof of successful migration.

DECLARATION

The authors should disclose any generative AI assistance in accordance with the target journal's policy. All technical claims, implementation status, metrics and references should be verified by the authors before submission.

REFERENCES

[1] GitHub, "REST API documentation." [Online]. Available: https://docs.github.com/en/rest. [Accessed: Oct. 2026].

[2] Octokit, "Octokit.js documentation." [Online]. Available: https://octokit.github.io/rest.js/. [Accessed: Oct. 2026].

[3] Google, "Gemini API documentation." [Online]. Available:

https://ai.google.dev/gemini-api/docs. [Accessed: Oct. 2026].

[4] React, "React documentation." [Online]. Available: https://react.dev/. [Accessed: Oct. 2026].

[5] TypeScript, "TypeScript documentation." [Online]. Available: https://www.typescriptlang.org/docs/. [Accessed: Oct. 2026].

[6] Node.js, "Node.js documentation." [Online]. Available: https://nodejs.org/docs/. [Accessed: Oct. 2026].

[7] Express, "Express web framework documentation." [Online]. Available: https://expressjs.com/. [Accessed: Oct. 2026].

[8] Prisma, "Prisma ORM documentation." [Online]. Available: https://www.prisma.io/docs. [Accessed: Oct. 2026].

[9] SQLite, "SQLite documentation." [Online]. Available: https://www.sqlite.org/docs.html. [Accessed: Oct. 2026].

[10] OWASP Foundation, "OWASP Top 10." [Online]. Available: https://owasp.org/www-project-top-ten/. [Accessed: Oct. 2026].

[11] Tailwind CSS, "Documentation." [Online]. Available: https://tailwindcss.com/docs. [Accessed: Oct. 2026].

[12] Vite, "Vite Guide." [Online]. Available: https://vite.dev/guide/. [Accessed: Oct. 2026].