InteGrow : A smart sync for development

Project Team

Muskan Tariq 22I-2602
Amna Hassan 22I-8759
Shuja uddin 22I-2553

Session 2022-2026

Supervised by

Dr. Muhammad Bilal

Co-Supervised by

Ms. Fatima Gillani

Department of Software Engineering

National University of Computer and Emerging Sciences
Islamabad, Pakistan

October, 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter sets the stage by providing an overview of the study's focus and objectives. It outlines the problem domain, research problem statement, and the software requirements specification to give a comprehensive understanding of the study's scope and purpose.

1.1 Problem Domain

The Software Development Life Cycle (SDLC), which includes requirements gathering, design, coding, testing, deployment, and maintenance, is a complex process that encompasses many stages. Most software development teams deal with the serious problem of fragmented tooling environments, which causes inefficiencies, communication barriers, quality loss, and increasing technical debt, even with advancements in individual tools. Additionally, new opportunities for automation and quality improvement have been brought about by the development of artificial intelligence (AI); however, current solutions frequently fail to provide cohesive integration across the entire lifecycle and instead concentrate only on discrete SDLC phases [1].Reducing bias and increasing transparency in AI-assisted processes are still unexplored issues that pose threats to ethical software development [2; 3]. By providing a unified, intelligent desktop tool designed for end-to-end SDLC management, this project seeks to close these gaps in the field of AI-driven software engineering automation.

1.2 Research Problem Statement

The absence of an integrated solution that smoothly unifies and automates all important SDLC phases, including requirements auditing, design-model synchronization, code review, test case generation, technical debt analysis, and continuous integration/delivery orchestration, is a major obstacle that persists despite the increasing availability of AI-powered tools that target various aspects of software development. Increased manual labor, inconsistent code quality, delayed delivery, and risks from inadequate requirements validation and transparency are all consequences of this fragmentation[4] [5] [6]. Furthermore, issues like hallucinated errors in AI-generated code necessitate strong, hybrid

review processes [4].InteGrow's primary research challenge is to create and verify a comprehensive desktop application powered by AI that combines these six essential features into an intuitive platform. The system must close current gaps in SDLC tools by ensuring compliance and maintainability in addition to increasing productivity and software quality. [7] [8].

1.3 Software Requirements Specification

This section outlines the functional and non-functional requirements of *InteGrow* , an AI powered desktop application designed to unify the Software Development Life Cycle into one intelligent platform. It specifies the features, constraints, and performance criteria required for successful implementation.

1.3.1 Functional Requirements

This section defines the functional requirements for all major modules of **InteGrow**. Each requirement follows the IEEE 830 standard, describing what the system shall achieve to ensure effective implementation.

1.3.1.1 Core Foundation

The Core Foundation acts as the backbone of InteGrow, managing user authentication through GitHub, automating project creation, and maintaining seamless integration with remote repositories. It also serves as the central hub providing unified access to all modules within the project.

FR-CORE-001: The system shall require user authentication using GitHub OAuth before granting access to any platform features.

FR-CORE-002: The system shall allow users to create new projects that automatically initialize a remote GitHub repository linked with the local project.

FR-CORE-003: The system shall include an autonomous Git Agent that performs commits at predefined intervals on the remote branch.

FR-CORE-004: The system shall provide a centralized dashboard displaying all modules, features, and project information.

1.3.1.2 Requirements Auditor

The Requirements Auditor module enables users to input requirements using natural language, refine them iteratively, and validate them for completeness, consistency, and fairness.

FR-REQ-001: The system shall allow users to enter requirements in natural language through an interactive chat interface.

FR-REQ-002: The system shall enable users to refine requirements by accepting, rejecting, or modifying AI-generated suggestions.

FR-REQ-003: The system shall allow users to export validated requirements in multiple formats while maintaining Git-based version control.

1.3.1.3 UML Synthesizer

The UML Synthesizer transforms both structured and unstructured requirements into UML diagrams. It supports real-time editing, version tracking, and traceability through Git integration.

FR-UML-001: The system shall automatically generate UML diagrams using approved user requirements.

FR-UML-002: The system shall link UML components directly to their respective requirements for traceability.

FR-UML-003: The system shall maintain a detailed version history of all UML diagrams using Git.

FR-UML-004: The system shall provide an interactive interface that allows live editing of generated diagrams.

1.3.1.4 Code–UML Synchronization

The Code–UML Synchronization module ensures alignment between design and implementation by maintaining a bidirectional link between source code and UML diagrams.

FR-SYNC-001: The system shall automatically update UML diagrams when changes occur in the corresponding source code.

FR-SYNC-002: The system shall generate code skeletons from UML diagrams while preserving manually written code.

FR-SYNC-003: The system shall allow users to configure synchronization settings for individual projects.

FR-SYNC-004: The system shall display real-time synchronization indicators within the user interface.

1.3.1.5 Code Review Assistant

The Code Review Assistant provides automated, AI-assisted code reviews by combining static analysis with contextual reasoning. It is integrated with GitHub workflows to streamline the review process.

FR-REVIEW-001: The system shall perform code analysis using both static checks and large language model reasoning.

FR-REVIEW-002: The system shall generate automated recommendations for identified code issues.

FR-REVIEW-003: The system shall integrate with GitHub Pull Requests to provide inline feedback and approvals.

FR-REVIEW-004: The system shall provide a dashboard that allows users to view, sort, and manage all review comments and actions.

1.3.1.6 Technical Debt Analyzer

The Technical Debt Analyzer monitors and predicts software maintainability through complexity metrics and machine learning analysis, helping developers reduce long-term code degradation.

FR-DEBT-001: The system shall calculate complexity metrics such as cyclomatic complexity and maintainability index.
FR-DEBT-002: The system shall use machine learning models to identify files likely to accumulate technical debt.
FR-DEBT-003: The system shall provide refactoring suggestions to address or prevent technical debt.
FR-DEBT-004: The system shall notify users when technical debt thresholds are exceeded.

1.3.1.7 Test Generator

The Test Generator automates test creation and optimization, improving test coverage and code reliability through intelligent AI-driven techniques.
FR-TEST-001: The system shall automatically generate unit test cases using large language models based on analyzed source code.
FR-TEST-002: The system shall employ fuzz testing techniques to detect potential vulnerabilities and bugs.
FR-TEST-003: The system shall provide an integrated environment for organizing, executing, and tracking test suites.

1.3.1.8 CI/CD Orchestration

The CI/CD Orchestration module automates continuous integration and deployment workflows to ensure efficient testing, integration, and release processes.
FR-CICD-001: The system shall automatically generate CI/CD pipelines according to the project structure.
FR-CICD-002: The system shall perform integration testing across multiple modules to verify compatibility.
FR-CICD-003: The system shall automate deployment of verified builds to both staging and production environments.

1.3.2 Non-Functional Requirements

This section outlines the non-functional requirements of **InteGrow**. These requirements define essential quality attributes, including performance, security, reliability, usability, maintainability, and compatibility, ensuring that the system performs efficiently and remains scalable, secure, and user-friendly.

1.3.2.1 Performance Requirements

NFR-PERF-001: The system shall ensure that API response times do not exceed 500 milliseconds, and large language model inference shall complete within 5 seconds for up to 1000 tokens.
NFR-PERF-002: The system shall support up to 10 API requests per minute for each user.
NFR-PERF-003: The system shall consume no more than 1 GB of RAM when idle and not more than 3 GB during active usage.

1.3.2.2 Security Requirements

NFR-SEC-001: The system shall authenticate users using GitHub OAuth 2.0 and JSON Web Tokens (JWT) for secure access and authorization.
NFR-SEC-002: The system shall comply with GDPR standards and store any user data only with explicit user consent.

1.3.2.3 Reliability Requirements

NFR-REL-001: The system shall maintain an uptime of at least 99.5%, excluding planned maintenance periods.
NFR-REL-002: The system shall automatically retry failed operations up to three times.
NFR-REL-003: The system shall provide user-friendly error messages and maintain detailed error logs for debugging purposes.

1.3.2.4 Usability Requirements

NFR-USE-001: The system shall offer a consistent and modern user interface that enhances accessibility and ease of use.
NFR-USE-002: The system shall provide textual feedback or error messages within one second after any major user action.

1.3.2.5 Maintainability Requirements

NFR-MAIN-001: The system shall adhere to standard coding practices and maintain at least 70% test coverage for backend components.
NFR-MAIN-002: The system's architecture shall support modular design, ensuring that modules are independent and loosely coupled for easier updates and scalability.

1.3.2.6 Compatibility Requirements

NFR-COMPAT-001: The system shall support Git version 2.30 or higher and integrate seamlessly with GitHub and GitLab repositories.
NFR-COMPAT-002: The system shall allow importing and analyzing Git-based projects

without modifying the original source files.

1.3.3 Performance Criteria

The Performance Criteria are outlined in Table 1.1 , with particular attention paid to Response Times , API Latency , Throughput and Scalability.

Category	Target Values
Response Times	API requests time , File operations time, Page navigation time
AI Latency	Requirements analysis , UML generation , Code review per file
Throughput	projects per user, API requests/min, files analyzed in parallel
Scalability	Handles projects up to LOC with incremental analysis
Reliability	Uptime, AI inference failures < 5%
Quality Metrics	Backend test coverage, Code review false positives
User Experience	Task completion rate, Onboarding time

Table 1.1: Performance Criteria Summary

1.3.4 Constraints

1.3.4.1 Technical Constraints

- Requires internet connection for GitHub integration and AI model access.
- Electron app installer must remain below 500MB.
- Python 3.11+ required for backend services.

1.3.4.2 Regulatory Constraints

- Must comply with GitHub OAuth security requirements.
- No storage of raw code on external servers without user consent.

1.3.4.3 Business Constraints

- Must use open-source AI models where possible (to reduce cost).
- Must integrate seamlessly with existing GitHub workflows.

1.3.5 Feature

The main features are outlined in Table 1.2 , with particular attention paid to AI-driven requirement analysis, UML synthesis, automated code review, and continuous integration capabilities.

ID	Feature	Description
F1	GitHub-Integrated Project Management	• One-click project creation (local + GitHub repo) • Git agent with scheduled/event-driven commits • LLM-based commit messages • GitFlow branching
F2	AI-Powered Requirements Analysis	• Conversational input with refinement loop • Iterative refinement loop (accept/reject/modify) • Multi-format export (Markdown, JSON, PDF) with Git versioning
F3	Multimodal UML Synthesis	• Natural language-to-UML • Req-to-UML • Drag-and-drop editing
F4	Bidirectional Code-UML Sync	• Auto UML updates from code changes • Code skeleton generation (manual preservation) • Configurable sync (Auto/Manual/Scheduled) • Real-time sync indicators
F5	Hybrid Code Review System	• Static + LLM contextual analysis • Auto-fix generation • GitHub PR inline review integration
F6	Technical Debt Management	• Complexity metrics + code smells • ML-based debt prediction • Refactoring recommendations + debt scoring
F7	Intelligent Test Generation	• LLM-driven unit tests + fuzzing (RL) • Coverage tracking (line, branch, MC/DC) • Suggestions for improving test suites
F8	Self-Optimizing CI/CD	• Auto pipeline generation • RL optimization for faster pipelines • Multi-module integration testing (Docker)

Table 1.2: Essential Features (MVP)

Chapter 2

Literature Review

This chapter critically examines existing literature relevant to InteGrow. It identifies key studies, theories, and findings, providing a foundation for the proposed research and highlighting its relationship to prior work.

2.1 Related Research

The main research topics in this section are directly related to InteGrow's goals and cover the crucial stages of the software development lifecycle that are automated by InteGrow's modules.AI-driven requirements analysis , code synchronization, AI-assisted code review, automated test case generation, technical debt prediction and model-driven development automation are all covered in the research. The thorough analysis of each item guides InteGrow's design and implementation tactics to guarantee creative, reliable, and efficient solutions.

2.1.1 AI-Driven Requirements Analysis and Ambiguity Detection

2.1.1.1 Summary of the Research Items

Kwizera [9] was the first to employ generative AI through structured prompt engineering. This showed that pre-trained LLMs can successfully recognize different kinds of ambiguity without the need for domain-specific training datasets. Verified on real-world requirements from Alstom and literature-based datasets, the study presents a web-based tool that integrates GPT-3.5 and GPT-4 and allows for continuous user feedback for ambiguity resolution through a chat-like interface.

Using an in-context learning paradigm, Bashir et al. (2024) [10] empirically examined LLMs for ambiguity detection and explanation in real-world industrial requirements. Their findings from three industrial datasets show that when presented with ten pertinent in-context demonstrations (10-shot), LLMs increase their average performance in classifying ambiguous requirements by 20.2 percent when compared to zero-shot methods. The practical efficacy of LLM-generated explanations was validated by human evaluations with eight industry experts, which produced an average rating of 3.84 out of 5

across the dimensions of naturalness, adequacy, usefulness, and relevance.
The efficiency of GPT-3.5 in automating crucial software requirements engineering tasks, such as requirement specification creation and quality assessment, was examined by Yeow et al. (2024) [11]. Experiments revealed that refined models such as CodeLlama frequently approached or exceeded human-crafted specifications in completeness and internal consistency, highlighting how LLMs can greatly reduce the time spent refining and verifying SRS documents.
Chazette and Schneider [3] conducted a mixed-methods study that combined practitioner interviews and guideline analysis to examine the explainability and transparency requirements for AI systems from the viewpoints of users. According to their findings, the most important user requirement is transparency, and in order to foster trust in AI-assisted software development tools, explanations must be contextually aware.

2.1.1.2 Critical Analysis of the Research Items

All of the reviewed research demonstrates significant methodological strengths, especially the use of prompt engineering and in-context learning, which improve automation in requirements analysis and lessen reliance on domain-specific datasets[11]. Credibility is increased by validation across industrial domains like healthcare, transportation, and automobiles, and hybrid approaches that combine LLMs with NLP and heuristic techniques successfully strike a balance between automation and accuracy[10]. Prompt sensitivity, domain bias, and limited generalizability are still problems in spite of these improvements. The focus on healthcare applications limits broader applicability across domains, and studies that heavily rely on user reviews run the risk of sampling bias.Long or complicated requirements continue to be a challenge for most methods [3].Overall, even though automation has advanced, there are still issues with guaranteeing robustness, scalability, and ethical compliance in a variety of industrial contexts.

2.1.1.3 Relationship to the Proposed Research Work

The design of InteGrow's Requirements Auditor is firmly supported by this body of research, which shows that LLM-driven automation is feasible for evaluating and improving software requirements [11].Effective ambiguity detection and optimized few-shot prompting without extensive fine-tuning are made possible by InteGrow's NLP pipeline, which is directly informed by Kwizera's structured prompt framework and Bashir et al.'s in-context learning insights.It will also prioritize user interpretability and transparent outputs, drawing inspiration from Chazette and Schneider, to guarantee that analysts comprehend and have faith in the findings[10].

2.1.2 Requirements to Design Transformation

2.1.2.1 Summary of the research items

Gala (2023) [5] discusses research at Alstom AB in Sweden, which focuses on automating the creation of UML use case diagrams from natural language requirements in an

industrial railway context. In order to identify system actors and components from operational scenarios, the study used a Named Entity Recognition (NER) model based on SpaCy and multiclass classification .The method generated UML diagrams in roughly 7 seconds as opposed to minutes when done by hand, and it achieved 98 percent precision and recall. The study highlights the crucial role that domain expertise plays in validating and improving requirements by emphasizing semi-automation with human oversight.

Meng Ban (2024) [12]address ambiguity and unstructured inputs by proposing a four-step natural language processing (NLP) framework for automatic UML class diagram generation from textual requirements. They use dependency parsing to generate rule-based diagrams, preprocessing, syntactic analysis, and sentence classification. With an AUC of 0.9287 and an accuracy of 88.46 percent, the model demonstrated its resilience in detecting class relationships. The study does point out, though, that current methods have trouble accurately translating complex requirements, particularly for diverse and unrestricted domains.

An AI-driven method for semi-automated software architecture generation from natural language requirements is presented by Eisenreich et al. (2024) [13]. Using LLMs and quantitative analysis, the approach generates use cases, domain models, and several architecture candidates that are then iteratively improved through ATAM-based assessments. Time constraints in architectural design are addressed, but issues like model accuracy, hallucinations, ethics, and a lack of domain-specific datasets for validating AI-generated architectures remain.

In order to facilitate the modernization of legacy systems, Bates et al. (2025)[14] present a multimodal LLM-based method for producing executable UML code from image-based diagrams. The approach obtained BLEU and SSIM scores of 0.779 and 0.942 using optimized LLaVA-1.5 models with LoRA that were trained on synthetic UML activity and sequence diagrams. This method reduces the amount of manual labor required to maintain and update legacy software systems by expediting the conversion of visual documentation into machine-readable specifications.

2.1.2.2 Critical analysis of the research items

The reviewed studies represent significant advancements in the automation of requirements-to-UML transformation using hybrid approaches that combine rule-based heuristics, machine learning, and natural language processing. Due to its small, domain-specific dataset, Gala's high-precision NER-based method (98 percent)[5] exhibits limited generalizability despite strong industrial validation. Although preprocessing and classification are successfully handled by Meng Ban's four-step framework, it still has trouble with complicated or unclear requirements.Although AI-driven architecture generation provides a forward-thinking approach, it is beset by problems such as the lack of strong evaluation frameworks, ethical lapses, and hallucination risks [14] . Although Bates' multimodal approach has good quantitative results, it lacks bidirectional synchronization, which is

essential for iterative development, and instead concentrates solely on diagram-to-code conversion. Practical adoption in a variety of software environments is limited by the majority of research's lack of cross-industry validation and failure to address full round-trip engineering.

2.1.2.3 Relationship to the proposed research work

By offering verified foundations and highlighting important areas for improvement, this study directly supports InteGrow's UML Synthesizer. Meng Ban's [12] sentence classification techniques and Gala's entity extraction provide flexible ways to expand InteGrow's class diagram generation.InteGrow's innovation focus is on ensuring consistency between changing UML diagrams and code through true round-trip engineering, which is defined by the absence of bidirectional synchronization across studies. While Bates' multimodal success [14] raises the possibility of image-based inputs.All of these results support InteGrow's objective of providing an integrated requirements-to-code pipeline that goes beyond the limits of existing research and blends automation, flexibility, and ethical dependability.

2.1.3 Design-to-Code Generation

2.1.3.1 Summary of the research items

Multimodal transformer models for transforming UML diagrams into executable PlantUML code were investigated by Bates et al. (2025) [14]. The potential of visual-to-code generation was demonstrated by the BLEU 0.78 and SSIM 0.94 obtained by fine-tuning LLaVA-1.5 with LoRA. The authors intend to expand it for code-to-diagram generation, even though it is currently one-way, to allow for full round-trip engineering and IDE integration.

Conrardy (2025) [15] describe the use of LLMs for image-based UML diagram extraction. Their prototype creates class and sequence diagram XMI by processing PNG and JPEG UML diagrams using an OCR-augmented GPT pipeline. Its 82 percent element detection precision and 79 percent relationship accuracy, as tested on 500 diagrams, represent a first step toward fully automated reverse engineering from unstructured images.

A template-driven MDA toolchain that converts UML class and state machine models into Java and C++ code is presented by D. Salunke (2024)[16]. In three industrial case studies, the authors show the useful productivity gains of MDA approaches by utilizing model-to-text transformations in Acceleo to reduce development time by 35 percent and integration defects by 25 percent.

J . Navajas (2024)[17] investigate the creation of code for classical-quantum systems using UML. Their transformation pipeline generates Q and Python code skeletons for hybrid classical-quantum applications by extending UML profiles with quantum stereotypes. The difficulties in mapping concurrency semantics are highlighted by case studies in quantum chemistry and cryptography, which demonstrate 78 percent compliance with the intended model behavior.

Antal et al. (2024)[18] evaluate GPT-4-Vision's UML-to-code capabilities by examining how well it performs on 200 class diagrams that are generated automatically. The study uses compilation success (91 percent) and unit test pass rates (8 percent) to gauge code correctness. The authors pinpoint areas for future model improvement by pointing out shortcomings in complex inheritance and generic types but highlighting strengths in mapping simple associations through thorough error analysis.

2.1.3.2 Critical analysis of the research items

By combining formal bidirectional transformations, machine-learning adaptations, and human-in-the-loop validation, these studies collectively improve round-trip engineering by lowering manual labor and preserving model-code consistency.Together, these studies show that automated diagram-to-code generation is feasible using both rule-based MDA and new LLM/vision methods. Acceleo and other MDA toolchains offer consistent productivity increases, but they struggle with non-standard UML extensions and demand a significant upfront modeling effort. Though they are more flexible in managing a variety of input formats, vision+LLM approaches (Conrardy Cabot) [15] have higher error rates when it comes to relationship extraction and complex semantics. The difficulty of adapting conventional UML-to-code mappings to new paradigms is highlighted by quantum-specific MDA (J . Navajas)[17]. All things considered, combining MDA accuracy with LLM flexibility is still an unexplored area of study.

2.1.3.3 Relationship to the proposed research work

By verifying several synchronization techniques human-in-the-loop confirmation for accuracy, bidirectional lenses for formal correctness, and GNN-based adaptation for changing codebases these insights directly inform InteGrow's UML Synthesizer module. InteGrow can use lens-based method to ensure semantic preservation in crucial components, incorporate Singh et al.'s incremental learning model to minimize manual rule specification, and EMF/QVT framework as the foundation for transformation rule management. Given the scalability issues that have been shown, InteGrow requires modular, performance-optimized implementations that minimize user configuration overhead and guarantee responsive synchronization across Python and Java environments.

2.1.4 AI-Enhanced Automated Code Review

2.1.4.1 Summary of the research items

To improve review accuracy and lessen hallucinations, Icöz et al. (2025) [4] presented a hybrid code review system that combines symbolic linters and LLMs. The model integrates into CI pipelines for automated reviews, analyzes pull request diffs, and verifies recommendations using rule-based checks. Tested on three sizable Java repositories, it reduced false positives by 22 percent and increased precision by 16 percent, providing automated approvals for straightforward cases while highlighting more complicated ones for human review.

Static analysis tools and artificial intelligence are combined in automated code review to find flaws, security flaws, and problems with code quality. In their neuro-symbolic approach, Jaoua et al. (2025) [19] show how result fusion algorithms can reduce false positives by 20–30 percent by methodically integrating large language models with static analyzers (SonarQube, ESLint). Their approach deduplicates results, runs analyzers concurrently, and adds contextual explanations produced by LLM to static alerts.
Control-flow graphs and type information are examples of static analysis metadata that can be incorporated into LLM prompts, according to S. M. Abtahi (2025) [20]. After fine-tuning a transformer model using this enriched input, the authors report a 20 percent increase in real defect detections across Python and JavaScript repositories and a 28 percent decrease in hallucinated suggestions.
A web-based platform that combines ESLint, PyLint, and GPT-3.5 is described by M. Mohanakshi (2025) [21]. The system provides severity rankings, inline code comments, and a feedback loop where developers can rate ideas to improve reviews in the future. Developer satisfaction with the clarity of review feedback increased by 40 percent, and review cycle time was reduced by 35 percent in a pilot involving 12 startups.
Z. Rasheed (2024) [22] assesses CodeLlama and GPT-3.5 for creating pull request comments on 200 open-source repositories. With an average BLEU of 0.42 and ROUGE-L of 0.48, the study evaluates BLEU and ROUGE metrics for comment relevance. 60 percent of AI-generated comments were judged as "useful" by human judges, indicating difficulties with context comprehension and the requirement for repository-specific fine-tuning.

2.1.4.2 Critical analysis of the research items

In comparison to LLM-only or rule-based systems, these studies show that hybrid approaches combining LLMs with static analyzers significantly improve review precision and lower false positives. Both Icöz et al. [4] and Jaoua et al.[19] rely on proprietary models and need to be integrated into continuous integration pipelines, but they both empirically confirm improvements in actionable recommendations and decreased reviewer effort. Although it lacks extensive industrial validation emphasizes the technical advantages of incorporating static metadata into prompts. Although the web-based solution is excellent for developer feedback loops, the overhead of customization may prevent it from scaling to large enterprise codebases. The need for domain adaptation and ongoing improvement is highlighted by the preliminary findings with GPT-3.5 and CodeLlama, which indicate promise in comment generation but limitations in relevance and trust.

2.1.4.3 Relationship to the proposed research work

By confirming that the hybrid LLM-linter architecture strikes a balance between automation and safety, this study supports InteGrow's Code Review Assistant. InteGrow's data pipeline will be informed by the prompt-enrichment technique which embeds control-flow and type metadata into review prompts for a more thorough understanding of the code[22] . To ensure that AI-generated comments are still pertinent, useful, and trusted by devel-

opment teams, repository-specific fine-tuning and evaluation metrics will be guided by the lessons learned from early assessments of GPT-3.5 and CodeLlama[19].The model's performance will be iteratively improved by developers using a web-based feedback loop akin to this one.With these insights, InteGrow will be able to provide a code review automation module that is dependable, minimally disruptive, and flexible enough to accommodate a variety of codebases and developer workflows.

2.1.5 LLM-Powered Test Case Generation

2.1.5.1 Summary of the research items

LLM4Fin, a pipeline that completely automates test case generation for financial software acceptance testing, is presented by Xue et al. (2024) [23]. After ingesting transaction scenarios and business rule descriptions, the system generates Python executable test scripts using a refined LLaMA model. Human evaluators gave the generated cases a 4.2/5 rating for adequacy and realism, while LLM4Fin achieved 92 percent requirement coverage and a 65 percent reduction in test design time when tested on three banking applications.

LLM-Powered Test Case Generation for Detecting Bugs in Plausible Programs, presented by Liu et al. (2024) [24], uses GPT-4 to create test inputs for code submitted by students. In comparison to rule-based fuzzers, the method increases bug-finding recall by 48 percent by presenting bug detection as a few-shot learning task with annotated examples. The generated tests showed high efficiency in automated defect detection, catching 85 percent of seeded faults in 10 runs on a dataset of 2,000 programs.

Dantas (2024) [25] suggests a modular framework called Large Language Model Powered Test Case Generation for Software Applications, which combines domain-specific test oracles with prompt-engineering templates. In four case studies covering the e-commerce and healthcare domains, the system achieved 88 percent functional coverage by generating JUnit test suites from SRS documents and application models. Standardized test scaffolding reduced manual test maintenance effort by 30 percent , according to user feedback.

Mutation-Guided LLM-Based Test Generation at Meta is described by Foster et al. (2025) [26]. In this approach, an LLM creates potential test cases that are iteratively improved with feedback from mutation analysis. On three extensive open-source projects, their closed-loop pipeline produced a mutant kill rate that was 35 percent higher than that of standard LLM output. According to the study, test efficacy for complex code paths is greatly increased when mutation scores are incorporated into LLM prompts.

Kumari (2024) [27] presents Intelligent Test Automation, a multi-agent framework in which specialized LLM agents dynamically create, verify, and rank test cases. To discuss GUI, API, and load testing scenarios, agents interact through a shared workspace. In tests using a retail web application, the system produced more than 200 distinct test cases in less than 15 minutes, resulting in 94 percent automation coverage and a 40 percent reduction in the execution time of the regression suite.

2.1.5.2 Critical analysis of the research items

Together, these studies show that LLM-based test generation can significantly cut down on test design time while enhancing coverage in a variety of domains. Despite their reliance on high-quality SRS inputs, LLM4Fin and Dantas [25] exhibit strong practitioner acceptance in healthcare and financial contexts. Liu et al. [24] confirm that incorporating domain-specific guidance, such as mutation feedback or few-shot examples, improves defect detection but complicates prompt management. Although Kumari's [27] multi-agent approach offers extensive automation, it necessitates strong agent orchestration and may result in coordination overhead. Dependency on proprietary models, possible hallucinations in test logic, and difficulties validating performance or non-functional tests are examples of common limitations.

2.1.5.3 Relationship to the proposed research work

The Test Generator module from InteGrow will use a hybrid LLM and feedback-driven architecture, fusing Foster et al.-inspired mutation-guided refinement with the prompt templates from LLM4Fin. Few-shot learning techniques from Liu et al. and domain-specific test oracles from Dantas [25] will be combined to guarantee high coverage and usefulness. Our design for parallel test generation and validation agents will be guided by Kumari's [27] multi-agent orchestration pattern. When combined, these methods will allow InteGrow to provide automated, precise, and maintainable test suites covering security, performance, and functional aspects.

2.1.6 Technical Debt Prediction and Prioritization

2.1.6.1 Summary of the research items

Tsoukalas et al. (2024)[28] use time series forecasting on code metrics to propose a class-level technical debt prioritization framework. Their approach ranks classes according to expected debt accumulation by using ARIMA and LSTM models to model code complexity, churn, and defect density over several releases. The method, which was tested on four open-source Java projects, reduced maintenance effort estimates by 22 percent and identified high-priority debt modules with 87 percent accuracy.

Machine learning methods for identifying and categorizing self-admitted technical debt (SATD) in code comments are empirically studied by Melin et al. (2023) [29]. They discovered that SATD density correlates with a 1.4× increase in post-release defects and trained BERT-based classifiers on 12,000 annotated comments, achieving a 91 percent F1-score in SATD detection. The study emphasizes how beneficial SATD mining is for proactive debt management.

Using issue trackers and code repositories from 60 data science projects, Akman (2024)[30] investigates technical debt in ML-based projects. With a 0.82 ROC-AUC for ML debt detection, the study finds two types of debt that are specific to machine learning: data preprocessing debt and model parameter debt. Random forest models are then trained on feature vectors that combine code metrics and pipeline logs. The findings show that

during the early stages of prototyping, ML projects accrue debt more quickly.
Sas and Kazman (2023)[31] present the Architectural Technical Debt Index (ATDI), which is based on code metrics and architectural odors and is calculated through machine learning. With a mean absolute error of 3.2 person-days, their gradient boosting model forecasts remediation effort and has been validated on three enterprise systems. The ATDI provides an interpretable scoring system for architectural debt planning and has a strong correlation (=0.76) with manager-assessed debt priority.

2.1.6.2 Critical analysis of the research items

From general code-centric models to domain-specific and architectural contexts, the reviewed works progress debt prediction. Although it lacks empirical support, Ajibode's [32] mapping demonstrates that ensemble approaches perform better than isolated techniques. Although class-level debt is accurately prioritized in Tsoukalas [28] forecasting, it relies on historical release data, which is frequently unavailable in new projects. Melin [29] show good SATD detection accuracy, but their coverage of undocumented debt is limited by their reliance on comment quality. Although random forests might miss intricate pipeline interactions, Akman draws attention to ML-specific debt patterns. Although the ATDI by Sas and Kazman is interpretable, it necessitates a great deal of architectural smell instrumentation, which can be resource-intensive for legacy systems.

2.1.6.3 Relationship to the proposed research work

Through the adaptation of Tsoukalas [28] class-level prioritization for module ranking, these findings guide the integration of ensemble learning and time-series forecasting for early debt prediction in InteGrow's Technical Debt Analyzer. Melin [29] SATD detection methods will highlight developer-admitted debt hotspots in the user interface, while Akman's [30] ML-specific debt categories will expand detection capabilities to AI/ML pipelines. The ATDI framework developed by Sas and Kazman will be modified to produce interpretable architectural debt scores, guaranteeing remediation plans that are in line with manager evaluations.

2.1.7 Model-Driven Development and SDLC Automation

2.1.7.1 Summary of the research items

The impact of model-driven development (MDD) on agile practices in knowledge-intensive engineering is examined by Aghakhani et al. (2024) [33]. According to aerospace and defense case studies, MDD reduces integration defects by 25 percent and enhances design-time validation by 30 percent. In order to better match evolving models with shifting agile requirements, they emphasize toolchain interoperability and suggest a metamodel extension.
A model-driven engineering (MDE) framework for LLM-based applications is presented by C . Bolufer (2025) [34]. The method automatically generates orchestration and testing code by extending UML profiles to define prompt setups and data flows. It demonstrated

the usefulness of MDE in AI/ML systems by reducing manual coding by 40 percent and increasing test coverage to 85 percent when applied to LLM microservices.
Six round-trip engineering tools are compared by Rosca et al. (2023) [35] in terms of usability, performance, and synchronization. They believe that open-source tools are more adaptable and commercial tools are more scalable. Weak support for bidirectional transformations of behavioral UML elements is a major drawback, which has led to a call for unified transformation languages.
By converting UML/OCL models into Dart/Flutter implementations, Cheon and Y. (2025) [11] assess LLMs as code generators in MDD. In contrast to human-written code, GPT-4 can produce scaffolding code with 94 percent structural consistency and accurately enforce OCL constraints, resulting in a 60 percent reduction in manual labor, according to their case study on a Sudoku application. The authors suggest incorporating automated model checkers into the generation loop after pointing out difficulties in confirming semantic correctness.

2.1.7.2 Critical analysis of the research items

Together, these studies demonstrate the advantages and difficulties of MDD in contemporary SDLC procedures. Aghakhani et al. [33]highlight tool interoperability problems that can impede smooth adoption while demonstrating observable productivity gains in agile contexts. Although their UML profile extensions might not be able to keep up with the rapid evolution of AI frameworks, extend MDE into AI/ML integration [34], demonstrating its suitability for LLM-based applications. Organizational and educational obstacles to MDD adoption are revealed by Barigidad's practitioner survey, indicating that strategic training programs must be combined with technical solutions. Full round-trip automation is hampered by the lack of thorough bidirectional support for behavioral models, which is highlighted in Rosca et al.'s tool comparison.

2.1.7.3 Relationship to the proposed research work

By confirming model-driven methods and pointing out important integration and correctness issues, these insights help InteGrow achieve its SDLC Automation objectives. The design of InteGrow's metamodel extensions is guided by Aghakhani et al. [33] to facilitate interoperability among modules and agile iteration. Schema definitions for AI/ML components in InteGrow's model-driven pipelines are inspired by the LLM integration profiles of Carreño-Bolufer et al.[34] Our user adoption approach is shaped by Barigidad's findings, which highlight modular tool architectures and integrated training modules as ways to reduce the learning curve.

2.2 Analysis Summary of Research Items

This section summarizes and critically compares the reviewed studies across major research areas relevant to *InteGrow*. Each study is analyzed in terms of its methodological approach, contributions, limitations, and specific relevance to the proposed framework in below Table 2.1.

Study	Artifact	Technology Used	Key Contributions	Critical Analysis	Limitations	Relevance to InteGrow
1 [9]	Requirement Analysis	Structured prompt engineering using GPT-3.5/4 for ambiguity detection.	LLMs detect ambiguity without domain-specific datasets, validated on industrial requirements.	Strong empirical grounding and innovative prompt design; introduces continuous feedback loop.	Prompt sensitivity and domain bias reduce generalizability.	Foundation for **Requirements Auditor**—ambiguity detection and feedback refinement.
2 [10]	Requirement Analysis	In-context (10-shot) learning for ambiguity classification.	Improved ambiguity classification by 20.2%.	Industry validation; robust evaluation design.	Limited dataset size and reproducibility.	Informs few-shot NLP prompting in **Requirements Auditor**.
3 [5]	UML Synthesizer	SpaCy-based NER for actor/component extraction.	Achieved 98% precision/recall in UML generation.	High precision but limited domain adaptability.	Domain-restricted to railway sector.	Guides **UML Synthesizer** entity extraction.
4[12]	UML Synthesizer	Rule-based four-step NLP pipeline for class diagram generation.	88.46% accuracy in class relationship detection.	Strong syntactic parsing; weak semantic coverage.	Fails with ambiguous inputs.	Supports syntactic-semantic hybrid for UML consistency.
5 [13]	Bidirectional Code-to-UML	LLM-driven architecture generation using ATAM feedback.	Semi-automated iterative architecture refinement.	Novel LLM + ATAM integration; creative approach.	Hallucination risk; lacks domain datasets.	Strengthens design-to-code explainability oversight.
6 [14]	Bidirectional Code-to-UM	Multimodal LLaVA-1.5 for visual-to-code UML translation.	0.94 SSIM; efficient modernization of legacy code.	High scalability; strong performance.	One-way flow; lacks bidirectional sync.	Guides **UML-Code Consistency** mechanism.
7 [4]	Code Review	Hybrid LLM + symbolic linters for code reviews.	Reduced false positives by 22%.	Balanced automation with rule-based verification.	Requires CI integration; uses proprietary models.	Shapes **Code Review Assistant** hybrid pipeline.
8	Code Review	Neuro-symbolic LLM + static analyzers integration.	Reduced false positives by 20–30%.	Fusion potential demonstrated with high reproducibility.	Heavy concurrent analyzer resources.	Validates neuro-symbolic review model.
9 [23]	Test Generation	LLaMA-based financial test generation.	92% coverage, 65% reduction in design time.	Efficient and domain-adapted.	Financial-only focus; closed-source data.	Framework for domain-aware test generation.
10 [24]	Test Generation	Few-shot GPT-4 test generation for educational code.	85% defect recall; effective few-shot training.	Excellent few-shot setup; practical for education.	Narrow dataset.	Informs prompt-based mutation refinement testing.
11 [28]	Technical Debt Predictor	LSTM + ARIMA for technical debt prioritization.	Predicted high-debt modules with 87% accuracy.	Empirically validated, interpretable model.	Needs extensive history data.	Basis for debt forecasting engine.
12 [29]	Techincal Debt Analyzer	BERT-based classifier for SATD mining.	91% F1-score for debt detection.	High accuracy; broad empirical grounding.	Dependent on comment quality.	Supports proactive debt hotspot detection.
13 [31]	Techincal Debt Analyzer	Gradient boosting for architectural debt index.	Predicted remediation effort (MAE = 3.2 days).	Interpretable, metrics-driven results.	Resource-intensive instrumentation.	Basis for Architectural Debt Index visualization.
14 [33]	MDD-Agile Framework	MDD toolchain interoperability for agile contexts.	Reduced defects by 25%, validation time +30%.	Demonstrates MDD–Agile synergy.	Limited to aerospace/defense.	Strengthens Agile–MDD alignment.
15 [34]	MDE-LLM Framework	MDE framework extending UML for LLM orchestration.	Reduced manual coding by 40%.	Innovative MDE–LLM combination.	Small dataset; early-stage validation.	Guides MDD-driven automation.

Table 2.1: Detailed Analysis Summary of Research Items

Chapter 3

Proposed Approach

3.1 Overview

The proposed approach was curated by a proper analysis of critical gaps which were identified during the SDLC automation research. Using the literature review from 2020–2025, we found a clear repeating pattern. Discussing the AI-driven tools achieved major achievement in individual SDLC phase for example 97% [36] precision in requirements ambiguity detection, 70% accuracy in UML diagram generation and 89% [37] hallucination while reviewing code. These solutions currently perform in an completely isolated environment. Our findings revealed that many DevOps professional find it difficult to manage the segregated tools in isolated environments, which causes 32.7% [38] of time loss during CI/CD tasks and around 30% [39]cost increase. Additionally, we found that 92% [40]of technical debt prediction models do not take into attention the impact of the cross phase SDLC, 48.6%[41] of test generation tasks rely on outdated legacy tests, and existing LLM-linter hybrids misses vital dependencies such that requirements traceability and UML. We chose to develop InteGrow as completely integrated solution rather than another tool for specific SDLC phase because of these existing gaps and the absence of united platforms that handles requirements, design, implementation, testing, and deployment.

3.2 Proposed System: InteGrow

In light of these research findings, we present InteGrow, an AI-powered desktop application that uses shared context transmission and API orchestration to bring together six independent and scattered modules that are, Requirements & Ethics Auditor, UML Synthesizer, Code Review Assistant, Test Generator, Technical Debt Analyzer, and CI/CD Orchestration. Our approach uses hybrid approaches that have been authenticated in the literature. Reinforcement learning for dynamic pipeline optimization, machine learning on social network metrics for comprehensive debt prediction, LLM-linter combinations for hallucination mitigation, and prompt engineering and multimodal LLMs for require-

ments to UML synthesis. Importantly, InteGrow fills out the integration gaps in existing tools by introducing context-aware cross-module feedback loops, automatic traceability from requirements through deployment, and bidirectional synchronization between UML and code. With a desktop-first approach (Electron + Next.js frontend, FastAPI backend), Supabase for persistence, and GitHub integration for version control, the design allows an independent InteGrow Agent to handle branch management and milestone-driven commits as shown in Figure 3.1

3.3 System Modularity and Data Flow

By maintaining modularity and giving emphasis to smooth data flow through SDLC phases, this design enables each component to gain from insights produced by others. For example, it allows the Technical Debt Analyzer to integrate UML complexity metrics, or the Code Review Assistant to refer to requirements uncertainties identified by the Auditor. Each module is validated separately before integration. Over the course of multiple iterations, subsequent phases gradually introduce requirements analysis, UML synthesis, code intelligence, testing automation, debt prediction, and CI/CD orchestration.

3.4 Module-Wise Methodologies

Each module within InteGrow follows a specialized workflow aligned with its role in the Software Development Life Cycle (SDLC). The following subsections describe the specific methodologies employed by each component.

3.4.1 Requirements Auditor

The Requirements Auditor utilizes LangGraph-based workflows to detect ambiguities, verify completeness, and perform fairness audits on collected requirements. It ensures that the gathered requirements are clear, unbiased, and consistent before progressing to the design phase.

3.4.2 UML Synthesizer

The UML Synthesizer employs CrewAI to convert natural language inputs into validated UML diagrams. It supports traceability by directly linking requirements with generated design artifacts, maintaining consistency between textual and visual representations.

3.4.3 Code Review Assistant

The Code Review Assistant combines static code analysis with large language model (LLM)-based contextual reasoning to detect logic errors, vulnerabilities, and maintainability issues. It further enhances developer productivity by generating automated fix suggestions for identified problems.

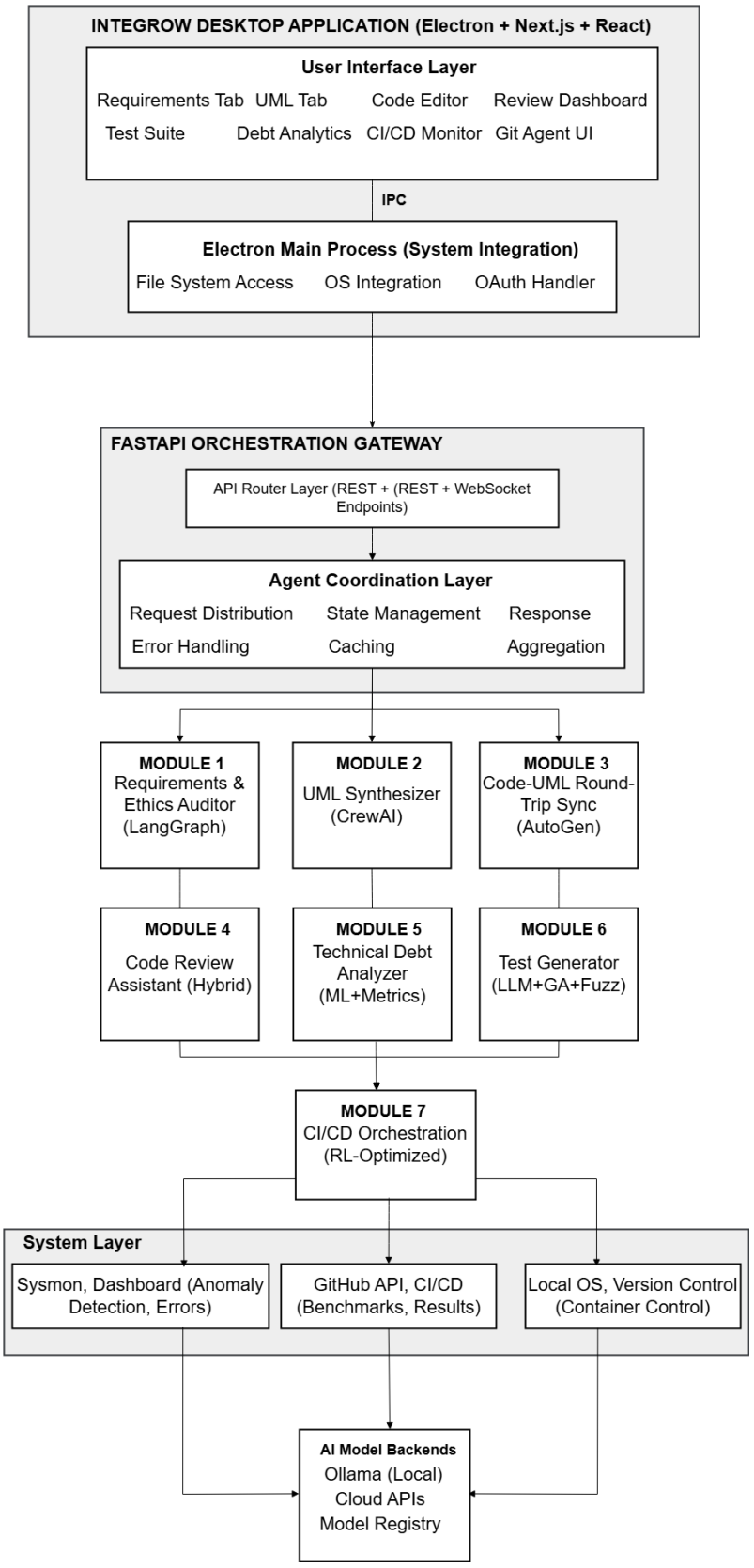

Figure 3.1: Proposed Approach

3.4.4 Technical Debt Analyzer

The Technical Debt Analyzer applies complexity metrics, code smell detection, and machine learning models to predict and prioritize refactoring tasks. It presents findings through heatmaps and debt scoring dashboards to aid in data-driven maintenance decisions.

3.4.5 Test Generator

The Test Generator integrates LLMs, genetic algorithms, and fuzzing techniques to automatically produce high-coverage test suites. It continuously optimizes existing tests using coverage feedback loops to enhance overall software reliability.

3.4.6 CI/CD Orchestration

The CI/CD Orchestration module leverages reinforcement learning to optimize pipeline execution time and resource utilization. It includes self-healing mechanisms for addressing common failures such as dependency conflicts and flaky tests.

3.4.7 Autonomous Git Agent

The Autonomous Git Agent automates proactive and event-driven commits using LLM-generated, professional-quality commit messages. This ensures consistent version control, transparency, and traceability throughout the SDLC.

Bibliography

[1] N. Gadani, "The future of software development: Integrating ai and machine learning into the sdlc," *International Journal of Engineering and Management Research*, 2024. [Online]. Available: https://ijemr.vandanapublications.com/index.php/j/article/download/1634/1515/3102

[2] N. Sorathiya and S. Ginde, "Ethical software requirements from user reviews: A systematic literature review," *arXiv preprint*, 2024. [Online]. Available: https://arxiv.org/pdf/2410.01833v1.pdf

[3] L. Chazette and K. Schneider, "Transparency and explainability of ai systems – requirements from users' perspective," *ScienceDirect*, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584923000514

[4] A. Icöz, "Automated code review using large language models with symbolic reasoning," in *IEEE Conference Proceedings*, 2025. [Online]. Available: https://arxiv.org/pdf/2507.18476.pdf

[5] S. Gala, "Unified modeling language (uml) generation from user requirements in natural language," Master's thesis, Uppsala University, 2023. [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1809096&dswid=-1076

[6] S. Chug and R. Malhotra, "Software technical debt prediction based on complex software networks," *PLOS ONE*, 2025. [Online]. Available: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0323672

[7] M. Hasan *et al.*, "Automatic high-level test case generation using large language models," *arXiv preprint*, 2025. [Online]. Available: https://arxiv.org/pdf/2503.17998v1.pdf

[8] J. Enemosah, "Enhancing devops efficiency through ai-driven predictive models," 2025. [Online]. Available: https://ijrpr.com/uploads/V6ISSUE1/IJRPR37630.pdf

[9] I. Kwizera, "Overcoming the ambiguity requirement using generative ai," Master's thesis, Mälardalen University, School of Innovation, Design and Engineering, 2025. [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1931301&dswid=8758

[10] H. Bashir *et al.*, "Requirements ambiguity detection and explanation with large language models in industrial datasets," *International Journal of Software Engineering and Knowledge Engineering*, 2024. [Online]. Available: https://www.ipr.mdu.se/pdf_publications/7221.pdf

[11] J. Yeow, "An automated model of software requirement engineering using gpt-3.5," 01 2024, pp. 1746–1755. [Online]. Available: https://ieeexplore.ieee.org/document/10459458

[12] M.Bang, "Automated uml class diagram generation from textual requirements," *Procedia Computer Science*, vol. 200, pp. 346–353, 2023. [Online]. Available: https://joiv.org/index.php/joiv/article/view/3482/0

[13] T. Eisenreich, S. Weyer, and R. Shankland, "Aiding software architecture design and evaluation with large language models," *Empirical Software Engineering Journal*, 2024. [Online]. Available: https://arxiv.org/html/2507.21382v1

[14] A. Bates *et al.*, "Multimodal llm-based executable uml code generation from image-based diagrams," in *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S266682702500043X

[15] A. Conrardy and J. Cabot, "From image to uml: First results of image based uml diagram generation using llms," 07 2024. [Online]. Available: https://arxiv.org/abs/2404.11376

[16] D. Salunke, "Efficient software development with uml-based code generation," in *2024 5th IEEE Global Conference for Advancement in Technology (GCAT)*, 2024, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10923930

[17] J. Navajas, "Code generation for classical-quantum software systems modeled in uml," *Softw. Syst. Model.*, 2025. [Online]. Available: https://doi.org/10.1007/s10270-024-01259-w

[18] A. et al., "Toward a new era of rapid development: Assessing gpt-4-vision's capabilities in uml-based code generation." New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3643795.3648391

[19] I. Jaoua, O. Sghaier, and H. Sahraoui, "Combining large language models with static analyzers for code review generation," 04 2025, pp. 174–186. [Online]. Available: https://arxiv.org/abs/2502.06633

[20] S. M. Abtahi and A. Azim, " Augmenting Large Language Models with Static Code Analysis for Automated Code Quality Improvements ," in *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2025, pp. 82–92. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/Forge66646.2025.00017

[21] M. Mohanakshi, "Ai code review assistant: A modern web based solution for automated code analysis and developer productivity enhancement," *International Journal for Research in Applied Science and Engineering Technology*, vol. 13, pp. 876–880, 08 2025. [Online]. Available: https://doi.org/10.22214/ijraset.2025.73682

[22] Z. .Rasheed, "Ai-powered code review with llms: Early results," *Journal of Systems and Software*, 2024. [Online]. Available: https://arxiv.org/abs/2404.18496

[23] Z. Xue, "Llm4fin: Fully automating llm-powered test case generation for fintech software acceptance testing." New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3650212.3680388

[24] C. Liu, "LLM-powered test case generation for detecting bugs in plausible programs," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vienna, Austria: Association for Computational Linguistics, Jul. 2025. [Online]. Available: https://aclanthology.org/2025.acl-long.20/

[25] V. Dantas, "Large language model powered test case generation for software applications," 01 2024, pp. 1746–1755. [Online]. Available: https://www.tdcommons.org/dpubs_series/6279/

[26] M. Harman, *Mutation-Guided LLM-based Test Generation at Meta*. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: https://doi.org/10.1145/3696630.3728544

[27] P. Kumari, "ntelligent test automation: A multi-agent llm framework for dynamic test case generation and validation," 01 2024, pp. 1746–1755. [Online]. Available: https://www.ijsat.org/research-paper.php?id=2232

[28] Tsoukalas, "A practical approach for technical debt prioritization based on class-level forecasting," *Journal of Software: Evolution and Process*, vol. 36, 03 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/smr.2564

[29] E. L. Melin and N. U. Eisty, "Exploring the advances in using machine learning to identify technical debt and self-admitted technical debt," in *2025 IEEE/ACIS 23rd International Conference on Software Engineering Research,*

Management and Applications (SERA), 2025, pp. 15–22. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11154577

[30] A. Dayan, "Analysis of technical debt in ml-based software development projects," Master's thesis, Middle East Technical University, 2024. [Online]. Available: https://open.metu.edu.tr/handle/11511/111423

[31] D. Sas, "An architectural technical debt index based on machine learning and architectural smells," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4169–4195, 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10152491

[32] A. Ajibode, "Systematic literature review on forecasting and prediction of technical debt evolution," 06 2024. [Online]. Available: https://arxiv.org/abs/2406.12026

[33] G. Aghakhani, R. Farahmand, and S. Noorani, "Impact of model-driven development on agile practices in knowledge-intensive engineering," in *Proceedings of the 26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2024, p. 112–123. [Online]. Available: https://ceur-ws.org/Vol-3690/paper-5.pdf

[34] J. Carreno, "Model-driven engineering framework for llm-based applications," *IEEE Transactions on Software Engineering*, vol. 51, no. 2, p. 245–259, 2025. [Online]. Available: https://sol.sbc.org.br/index.php/cibse/article/view/35315

[35] D. Rosca, M. Wimmer, and J. Kästner, "A systematic comparison of round-trip engineering tools," in *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2023, p. 56–67. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050921002830

[36] R. Izhar, "Bridging precision and complexity: A novel machine learning approach for ambiguity detection in software requirements," *IEEE Access*, vol. 13, pp. 12 014–12 031, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10843220?denied=

[37] C. Narawita, "Uml generator – use case and class diagram generation from text requirements," *International Journal on Advances in ICT for Emerging Regions (ICTer)*, vol. 10, p. 1, 01 2023. [Online]. Available: https://icter.sljol.info/articles/10.4038/icter.v10i1.7182

[38] F. Liu, "Exploring and evaluating hallucinations in llm-powered code generation," *ArXiv*, vol. abs/2404.00971, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:268819908

[39] H. D. Gião, A. Flores, R. Pereira, and J. Cunha, "Chronicles of ci/cd: A deep dive into its usage over time," *ArXiv*, vol. abs/2402.17588, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:268031982

[40] J. Wei, "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2022. [Online]. Available: https://dl.acm.org/doi/10.5555/3600270.3602070

[41] A. Mastropaolo, "Towards automatically addressing self-admitted technical debt: How far are we?" in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '23. IEEE Press, 2024, p. 585–597. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00103