Agentic Digital Twin for Software Engineering (ADT-SE): A Multi-Agent AI Framework for Autonomous Understanding and Evolution of Software Repositories

Dr. Shagufta Sheikh[1], Sanwar Bendsure[2], Yash Bhandari[3], Deven Katariya[4], Gulab Jangid[5]

[1]Dr. Shagufta Sheikh (Project Guide / Mentor), Dept. of Artificial Intelligence & Data Science, AISSMS Institute of Information Technology, Pune, India

[2]Sanwar Bendsure, Dept. of Artificial Intelligence & Data Science, AISSMS Institute of Information Technology, Pune, India | sanwarbendsure60@gmail.com

[3]Yash Bhandari, Dept. of Artificial Intelligence & Data Science, AISSMS Institute of Information Technology, Pune, India | yashhhbhandari4@gmail.com

[4]Deven Katariya, Dept. of Artificial Intelligence & Data Science, AISSMS Institute of Information Technology, Pune, India | devenkatariya2005@gmail.com

[5]Gulab Jangid, Dept. of Artificial Intelligence & Data Science, AISSMS Institute of Information Technology, Pune, India | gulabjangid309@gmail.com

Abstract – Modern software repositories couple architecture, database, backend, frontend, API, test, CI/CD and infrastructure artefacts so tightly that no single developer or static tool can keep a complete and current understanding of them. Existing AI coding assistants and single-agent software-engineering (SWE) frameworks perform well within one file or one task, but they keep no persistent model of the repository, so every session starts from a partial and quickly outdated view. This paper presents the Agentic Digital Twin for Software Engineering (ADT-SE), a framework that clones a target GitHub repository and builds a continuously synchronised, queryable digital twin from it using a multi-language Abstract Syntax Tree (AST) parser, a NetworkX code property graph, and a ChromaDB semantic index. A seven-agent Software Development Life Cycle (SDLC) orchestrator implemented in LangGraph — comprising a Product Manager, System Architect, Security Engineer, Full-Stack Developer, QA Tester, Staff Reviewer, and DevOps Engineer — reasons over this twin through a shared typed state object, while a context builder grounds every prompt in graph-neighbourhood and retrieved code context. A bounded self-healing loop returns failing or rejected patches to the Developer agent; accepted changes are committed through a Git workspace manager, proposed on GitHub as pull requests, and re-indexed into the twin, closing the loop between the codebase and its model. A FastAPI gateway streams agent trajectories over server-sent events to a React + Vite dashboard that provides graph exploration, semantic search, and an AST-aware diff and merge studio for human review. This paper describes the architecture, the agent workflow and state model, staged-autonomy governance design, and an evaluation protocol based on SWE-bench-style tasks, seeded vulnerability injection and DevOps Research and Assessment (DORA) metrics against single-agent and manual baselines.

Key Words: Digital Twin, Agentic AI, Multi-Agent Systems, Large Language Models, LangGraph, Code Property Graph, Retrieval-Augmented Generation, Self-Healing Pipeline, Software Engineering Automation, DevOps.

1. INTRODUCTION

Software systems have grown rapidly in scale and interdependency. A single repository typically spans several services, data stores, Application Programming Interface (API) contracts, test suites and deployment pipelines that must remain mutually consistent as the code evolves. Developers routinely lose time re-establishing context that already existed somewhere in the repository — in an outdated architecture diagram, a stale README or a colleague's memory — because no persistent, machine-readable model of the system as a whole is maintained.

Artificial Intelligence (AI) coding assistants have made individual tasks faster, but they do not remove this problem. Inline completion tools and chat-based copilots are highly capable within a single file, yet they hold no lasting model of the repository, so each interaction begins from a partial and quickly outdated view. Single-agent SWE frameworks give one language-model agent tool access to browse a repository, edit files and run tests, but they work sequentially on one task and do not divide responsibility the way a real engineering team divides it across architecture, development, testing, review, security and operations.

Two complementary ideas point towards a solution. The first is the digital twin: a live, structured representation of a system that supports monitoring, prediction and process improvement. Kimmel et al. [1] argue that this paradigm should be extended from physical and runtime systems to software engineering processes, and Nechesov et al. [2] show that a twin can serve as the shared substrate on which societies of autonomous AI agents act. The second is agentic AI, in which several role-specialised language-model agents collaborate under an orchestration layer. Recent work applies this idea to the software lifecycle, from agent roles aligned with agile practice [4] to autonomous decision-making between commit and production [3]. What is still missing is a system that joins the two: a persistent twin of the whole repository acted upon, and kept current, by a team of agents.

This paper presents ADT-SE, the Agentic Digital Twin for Software Engineering. Given a GitHub repository, ADT-SE builds a cognitive digital twin consisting of parsed code structure, a code property graph and a semantic vector index. A seven-agent LangGraph orchestrator then performs requirement analysis, architectural planning, security screening, implementation, testing, review and delivery over that twin. Every accepted change is committed, proposed as a pull request, and re-indexed, so the twin and the codebase evolve together under human-auditable supervision instead of through one-off, human-triggered automation.

The main contributions of this work are:

- A repository-wide cognitive digital twin that fuses AST-derived structure, a NetworkX code property graph and a ChromaDB semantic index, with a context builder that combines graph and vector evidence into grounded prompts.
- A seven-agent SDLC orchestrator in LangGraph that mirrors an engineering team and coordinates through a shared, typed state object.
- A bounded self-healing loop, together with Git-based delivery and incremental twin re-indexing, that forms a self-evolving feedback cycle.
- A human-oversight and staged-autonomy design, realised through an AST-aware diff and merge studio and pull-request delivery, with a policy-as-code trust-tier model for expanding autonomy safely.
- An evaluation protocol that uses SWE-bench-style tasks, seeded vulnerability injection and DORA metrics against single-agent and manual baselines.

2. LITERATURE REVIEW AND RELATED WORK

Related systems were reviewed along three capabilities central to ADT-SE: (i) whether the system maintains a persistent, structured model of a software repository; (ii) whether several specialised agents collaborate rather than one generic agent acting alone; and (iii) whether a guardrail or governance mechanism constrains agent autonomy before an action is applied.

2.1 Digital Twins for Software Engineering

Kimmel et al. [1] propose extending the digital-twin paradigm to software engineering processes themselves, arguing that a live, structured model of a codebase's state can support monitoring, prediction and process improvement. This establishes the conceptual case for treating a repository as a continuously modelled living system. However, the proposed twin is passive and observational: no mechanism lets it drive code changes, testing or deployment decisions. Nechesov et al. [2] extend digital twins into virtual cities inhabited by societies of autonomous agents that collaborate, negotiate and act in a shared simulated environment. This shows that a twin can underpin coordinated multi-agent behaviour, but the work does not address software-engineering-specific concerns such as code review, test generation, CI/CD gating or guardrails for production-bound autonomy.

2.2 Multi-Agent and AI-Augmented SDLC Automation

Baqar et al. [3] study AI-augmented Continuous Integration/Continuous Delivery (CI/CD) pipelines in which autonomous decisions are taken between code commit and production. Their focus is the deployment stage rather than the full cycle of understanding, testing, reviewing and evolving a codebase. ALMAS [4] is the closest system to the agent design proposed here: it aligns Large Language Model (LLM) agents with agile roles to carry out SDLC tasks end-to-end alongside human developers. ALMAS does not, however, build or maintain a persistent twin of the repository, and it does not describe a staged-autonomy guardrail. Gogineni [5] presents a general framework in which heterogeneous LLM agents collaborate through optimised knowledge retrieval and communication, identifying coherent

reasoning and factual accuracy across agents as open challenges. The framework is domain-agnostic and is not evaluated on software-engineering tasks.

2.3 Governance and Repository-Level Grounding

Tong et al. [6] review emerging IEEE standards for governing agentic systems, which informs the policy-as-code, trust-tiered autonomy layer of ADT-SE. For grounding, Liu et al. [7] show with GraphCoder that coarse-to-fine retrieval over a code context graph substantially improves repository-level code completion. This motivates the decision to combine graph structure with semantic retrieval instead of relying on embeddings alone.

2.4 Single-Agent SWE Frameworks and Benchmarks

SWE-agent [8] shows that a language-model agent equipped with a purpose-built agent-computer interface can navigate a repository, edit files and run tests to resolve real issues. SWE-bench [9] provides a benchmark of real GitHub issues paired with held-out tests and has become the standard for comparing such systems. These systems are task-scoped: each run starts from a repository snapshot with no persistent, shared model, and a single agent plays every role. ADT-SE therefore uses a SWE-agent-style system as the single-agent baseline in evaluation.

2.5 Comparative Analysis and Research Gap

Table 1 summarises the reviewed systems against five capabilities. Digital-twin work provides no embedded agentic layer; agentic SDLC frameworks maintain no persistent twin; and AI-augmented CI/CD addresses mainly the deployment stage. No reviewed system combines a repository-wide twin, a coordinated society of role-specialised agents, a guardrail model that stages autonomy, and a loop in which every accepted action updates the twin that informs the next decision. ADT-SE addresses this gap.

Table 1: Comparison of Related Systems and ADT-SE

System	Digital Twin	Multi-Agent	Guardrails	CI/CD	Self-Evolving
[1] SE Digital Twins (ICSE-NIER 2025)	✓	✗	✗	✗	✗
[2] Virtual Cities AI Societies (IEEE Access 2025)	✓	✓	Partial	✗	Partial
[3] AI-Augmented CI/CD (FLLM 2025)	✗	✗	Partial	✓	✗
[4] ALMAS (ASEW 2025)	✗	✓	✗	Partial	✗
[5] LLM-MAS Framework (AIRC 2025)	✗	✓	✗	✗	✗
ADT-SE (Proposed)	✓	✓	✓	✓	✓

3. SYSTEM ARCHITECTURE

ADT-SE is organised into four cooperating subsystems: a presentation layer, a backend Application Programming Interface (API) gateway, a cognitive digital twin core, and a seven-agent SDLC orchestrator. A set of external services — a large language model, the GitHub API and a local cloned workspace — completes the system. The presentation layer and the gateway communicate over HTTP REST/JSON, with a server-sent-events (SSE) channel that streams live agent activity back to the browser. The gateway invokes the orchestrator, which obtains grounded context from the twin core and writes its results back to the cloned workspace.

3.1 Presentation Layer

The client is a single-page application (SPA) built with React and Vite. It contains six modules: a repository and file-tree explorer; a Cytoscape.js view of the code property graph (TwinView); a multi-agent pipeline panel with a live trajectory stream (AgentPanel); an AST and line-diff reviewer with a merge studio (DiffViewer); a Git and GitHub control panel (GitHubHub); and semantic code search. Together they let a developer inspect the twin, watch each agent's reasoning and output as it happens, and decide whether to accept a proposed change.

3.2 Backend API Gateway

The gateway is a FastAPI service that exposes four route groups: TwinRoutes (/api/twin/*) for building and querying the twin; AgentRoutes (/api/agents/*) for running and inspecting agents; StreamRoutes (/api/agents/run-stream) for SSE

streaming of the pipeline; and ExplainerRoutes (/api/explainer/*) for natural-language explanations of code and graph elements. Long-running pipeline executions use SSE so that each agent step is visible to the user as soon as it is produced.

3.3 Cognitive Digital Twin Core

The twin core converts a cloned repository into a structured, queryable model. Its components, in processing order, are: (i) a Repository Scanner that walks the cloned workspace and registers source, configuration and documentation files; (ii) an In-Memory File Store and Vector Index that holds file contents and metadata; (iii) a Multi-Language AST Parser that extracts classes, functions, imports and schema definitions from Python, JavaScript/TypeScript and SQL files; (iv) a Code Property Graph assembled with NetworkX [12], whose nodes are files, classes, functions and tables and whose edges capture containment, import and reference relations; (v) a ChromaDB ephemeral client that stores embeddings for Retrieval-Augmented Generation (RAG) [10]; (vi) a Context Builder that combines semantic hits with graph neighbours to produce grounded prompts for agents; and (vii) a Git Workspace Manager that clones, branches, diffs and commits through GitPython.

Cross-component links make the twin more than a document store: a query can start from a semantic match, move along graph edges to dependent files, and thereby estimate the blast radius of a proposed change before it is made.

3.4 Seven-Agent SDLC Orchestrator

The orchestrator is a LangGraph state graph. Seven role-specialised agents read from and write to a shared CentralState object — a typed dictionary — holding the requirement, twin context, plan, patch, test report, review verdict and retry counter. Because every agent works on the same typed state, hand-offs are explicit and the full trajectory can be logged and replayed. Table 2 summarises each agent's role.

Table 2: Agents in the ADT-SE Orchestrator

#	Agent	Primary Responsibility
1	Product Manager	Analyses the task/issue, produces structured requirements and acceptance criteria.
2	System Architect	Maps requirements onto the twin, identifies affected files and designs the change plan; flags structural drift.
3	Security Engineer	Screens the plan and proposed changes for insecure patterns and vulnerable dependencies; findings become constraints in shared state.
4	Full-Stack Developer	Generates a repository-compatible patch from the plan and the grounded context.
5	QA Tester	Validates the patch via syntax checks and generated tests; reports failures back into the self-healing loop.
6	Staff Reviewer	Reviews the diff against requirements, coding standards and security findings; returns a verdict with feedback.
7	DevOps Engineer	Commits the change, generates CI/CD configuration and opens the pull request on GitHub.

3.5 External Services and Integrations

All agents obtain language-model completions from the Google Gemini API [13] (gemini-2.5-flash, with gemini-1.5-flash as an alternative). The GitHub REST API v3 provides forking, commits, pull requests and authentication. The local cloned workspace — a filesystem repository managed through Git — is where patches are applied and validated before being pushed to GitHub.

4. METHODOLOGY

4.1 Twin Construction Pipeline

Twin construction follows a deterministic pipeline: the repository is cloned; the scanner enumerates its files; each file is stored and parsed; parsed entities are linked into the property graph; file contents are chunked and embedded into the vector index; and the context builder is bound to both stores. The pipeline is designed to be incremental — when an accepted change modifies a subset of files, only those files are re-parsed and re-embedded, keeping re-index latency proportional to the size of the change rather than the size of the repository.

4.2 Agent Workflow and Self-Healing Loop

The Product Manager, System Architect and Security Engineer agents run once and enrich the shared state sequentially. The Full-Stack Developer then produces a patch, which the QA Tester validates. If validation fails, a conditional edge returns control to the Developer together with the failure report. If validation succeeds, the Staff Reviewer evaluates the patch, and a RetryDecision node applies the following rule:

$$\text{next} = \text{Developer} \quad \text{if} \quad (\text{fail} \vee \text{reject}) \quad \wedge \quad r < 2, \quad \text{else} \quad \text{DevOps}$$

where r is the retry counter stored in the shared state. Each pass through the loop gives the Developer the previous failure or review feedback as additional context — hence the term self-healing loop. The counter bounds cost and prevents non-terminating cycles; once it is exhausted the pipeline moves on to the DevOps agent, and the remaining feedback stays visible to the human reviewer in the pull request.

4.3 Delivery and Twin Synchronisation

The DevOps agent commits the validated patch on a working branch through the Git workspace manager, generates CI/CD configuration where required, and uses the GitHub API to fork the repository and open a pull request. After the change, the affected files are re-parsed and re-embedded, and the property graph and vector index are updated incrementally. The next task therefore starts from a twin that reflects the repository as it now stands, rather than a stale snapshot.

4.4 Trajectory Streaming and Observability

Every agent step emits an event containing the agent name, a summary of its input, its output and timing. The StreamRoutes endpoint forwards these events over SSE to the AgentPanel, and the same trajectory is logged so that each decision can be audited after the fact. This observability allows a human to verify why a patch was produced, not only whether it passes tests.

5. HUMAN OVERSIGHT AND STAGED AUTONOMY

Because agents can propose changes to a real codebase, autonomy must be earned rather than assumed. ADT-SE applies three complementary mechanisms.

Reviewable Changes. Every change is delivered as a diff that a human can inspect in the DiffViewer, which provides AST-level and line-level views and a merge studio for accepting or editing individual hunks. Changes reach the upstream repository only as pull requests, so the final merge decision stays with the developer.

Trust Tiers. Following governance guidance in [6], each agent role and action type is assigned a tier through machine-readable policy rather than hard-coded logic. At the Advisory tier an agent only reports findings. At the Co-pilot tier it prepares an action — such as a pull request — that waits for explicit human approval. At the Autonomous tier it may act within pre-approved policy bounds, with the action fully logged. Every agent starts at the Advisory tier on a new repository. Promotion is proposed from accuracy observed over a rolling window of past decisions and must be confirmed by a maintainer, and any tier is reverted immediately if a proposal is rejected or a deployed change causes a regression.

Auditability. The logged trajectory records every agent decision, giving a complete trail for post-incident review. In the current prototype, all repository-changing actions are delivered through fork and pull-request workflows that require an explicit human merge, which corresponds to the Co-pilot tier. Automated tier promotion is specified as policy-as-code and is part of the planned extension.

6. IMPLEMENTATION

ADT-SE is implemented in Python and JavaScript. The cognitive twin is held in process memory and rebuilt for each session, which keeps the prototype simple and the deployment lightweight. Table 3 lists the technology stack by layer.

Table 3: Technology Stack

Layer	Technologies
Client	React, Vite, Cytoscape.js, SSE client
API Gateway	Python, FastAPI (REST/JSON and SSE)
Twin Core	Multi-language AST parser (Python, JS/TS, SQL), NetworkX, ChromaDB (ephemeral), GitPython

Layer	Technologies
Orchestration	LangGraph state graph, typed shared state, MCP tool access
LLM	Google Gemini (gemini-2.5-flash / gemini-1.5-flash)
Integrations	GitHub REST API v3, Git, Docker

7. EVALUATION PROTOCOL

7.1 Datasets and Benchmarks

The framework is evaluated using the following datasets:

- A corpus of open-source GitHub repositories in Python, JavaScript/TypeScript and Java, ranging from small single-service projects to larger multi-module repositories, to validate twin construction accuracy and end-to-end agent performance.
- SWE-bench / SWE-bench-Lite-style issue-resolution tasks [9] to compare the Developer and Tester agents against a single-agent baseline.
- Seeded bug-injection and OWASP-aligned vulnerability-injection datasets to evaluate the Security Engineer agent where labelled real vulnerabilities are scarce.
- CI/CD pipeline history (GitHub Actions logs) from selected repositories to establish DORA baselines before and after ADT-SE integration.

7.2 Baselines and Protocol

The evaluation follows a controlled three-way comparison on the same repositories and tasks: (a) a manual baseline in which a developer performs the task unaided; (b) a single-agent baseline — a SWE-agent-style tool [8] with tool access but without a twin or role specialisation; and (c) the full ADT-SE pipeline. Each issue-resolution task is recorded as resolved, partially resolved, or unresolved, together with the number of iterations, wall-clock time, and whether the patch passes the held-out test suite.

Where a repository has CI history, DORA metrics are computed retrospectively over the pre-ADT-SE commit history and prospectively over a trial period after ADT-SE integration. Ablations disable, in turn, the twin context, the Security agent, and the self-healing loop, to isolate the contribution of each component.

7.3 Evaluation Metrics

Table 4: Evaluation Dimensions and Metrics

Dimension	Metrics
Twin Quality	Precision and recall of extracted dependency edges against manually verified sample; re-index latency.
Task Success	Resolve rate; patch passes held-out tests; number of iterations; wall-clock time.
Code Quality	Test coverage; defect escape rate.
Security	Precision P = TP/(TP+FP) and Recall R = TP/(TP+FN) on seeded vulnerabilities.
Delivery (DORA)	Lead time for changes; deployment frequency; change-failure rate; Mean Time to Recovery (MTTR).

Security precision and recall are reported separately because false positives erode developer trust, whereas false negatives defeat the purpose of the agent.

8. DISCUSSION AND LIMITATIONS

The persistent twin is expected to help most on tasks where the relevant code is spread across files that a single-agent system would not open, and the role split and retry loop are expected to reduce defects that reach review. These are hypotheses to be confirmed by the ablation protocol described in Section 7.2.

Several limitations apply. First, output quality depends on a hosted LLM, so results may vary between runs and with model versions, and both cost and latency grow with each retry iteration. Second, the current parser covers Python,

JavaScript/TypeScript and SQL; repositories in other languages receive only semantic, not structural, support. Third, the ephemeral in-memory index is rebuilt per session, which limits scalability for very large repositories and long-lived twins; a persistent store such as PostgreSQL with a vector extension is a natural next step. Fourth, repository content is untrusted input and may contain prompt-injection text, so agent outputs must remain behind human review until stronger sandboxing is available. Finally, public benchmarks may overlap with model training data, which can inflate measured resolve rates.

9. CONCLUSION AND FUTURE SCOPE

This paper presented ADT-SE, a framework that maintains a live digital twin of a software repository and lets a seven-agent LangGraph orchestrator understand, test, review, secure and evolve that repository under human supervision. The combination of a graph-and-vector twin, a shared-state agent society, a bounded self-healing loop, and pull-request delivery with re-indexing closes the loop between code and model — which existing AI coding assistants, single-agent frameworks and passive twins each fail to do.

Future work includes: a dedicated Documentation agent and a UX/Accessibility agent for broader lifecycle coverage; twins that span several repositories and microservices for enterprise-scale systems; persistent twin storage using PostgreSQL with a vector extension; reinforcement learning from accepted and rejected decisions to improve agent policies over time; formal verification hooks and stronger sandboxing for fully autonomous production deployment; and packaging ADT-SE as a plug-and-play DevOps add-on compatible with GitHub, GitLab and Bitbucket workflows.

ACKNOWLEDGEMENT

The authors sincerely thank their project guide, Dr. Shagufta Sheikh, for her invaluable guidance, mentorship, and encouragement throughout this work. The authors also thank the Department of Artificial Intelligence and Data Science, AISSMS Institute of Information Technology, Pune, for providing the facilities and academic environment that made this research possible.

REFERENCES

[1] R. Kimmel, J. Michael, A. Wortmann, and J. Zhang, "Digital Twins for Software Engineering Processes," in Proc. 2025 IEEE/ACM 47th Int. Conf. on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), Ottawa, Canada, 2025, doi: 10.1109/ICSE-NIER66352.2025.00009.
[2] A. Nechesov, I. Dorokhov, and J. Ruponen, "Virtual Cities: From Digital Twins to Autonomous AI Societies," IEEE Access, vol. 13, pp. 13866–13903, 2025, doi: 10.1109/ACCESS.2025.3531222.
[3] M. Baqar, S. Naqvi, and R. Khanda, "AI-Augmented CI/CD Pipelines: From Code Commit to Production with Autonomous Decisions," in Proc. 2025 3rd Int. Conf. on Foundation and Large Language Models (FLLM), 2025, doi: 10.1109/FLLM67465.2025.11391007.
[4] V. Tawosi, K. Ramani, S. Alamir, and X. Liu, "ALMAS: an Autonomous LLM-based Multi-Agent Software Engineering Framework," in Proc. 2025 40th IEEE/ACM Int. Conf. on Automated Software Engineering Workshops (ASEW), 2025, pp. 287–290.
[5] V. Gogineni, "LLM-Powered Multi-Agent Systems: A Technical Framework for Collaborative Intelligence Through Optimized Knowledge Retrieval and Communication," in Proc. 2025 6th Int. Conf. on Artificial Intelligence, Robotics and Control (AIRC), 2025, pp. 452–456, doi: 10.1109/AIRC64931.2025.11077480.
[6] R. J. Tong, H. Li, S. Raghavan, et al., "IEEE AI Standards for Agentic Systems," in Proc. 2025 IEEE Conf. on Artificial Intelligence (CAI), 2025, pp. 1603–1609, doi: 10.1109/CAI64502.2025.00269.
[7] W. Liu, A. Yu, D. Zan, B. Shen, W. Zhang, H. Zhao, Z. Jin, and Q. Wang, "GraphCoder: Enhancing Repository-Level Code Completion via Coarse-to-Fine Retrieval Based on Code Context Graph," in Proc. 2024 39th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE), 2024, pp. 570–581, doi: 10.1145/3691620.3695054.
[8] J. Yang, C. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," in Proc. 38th Conf. on Neural Information Processing Systems (NeurIPS), 2024.
[9] C. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" in Proc. 12th Int. Conf. on Learning Representations (ICLR), 2024.
[10] P. Lewis, E. Perez, A. Piktus, et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in Proc. 34th Conf. on Neural Information Processing Systems (NeurIPS), 2020.
[11] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in Proc. 2014 IEEE Symp. on Security and Privacy, pp. 590–604, 2014.
[12] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX," in Proc. 7th Python in Science Conf. (SciPy), 2008, pp. 11–15.
[13] Google, "Gemini API Documentation," 2024. [Online]. Available: https://ai.google.dev/docs
[14] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. IT Revolution Press, 2018.