ALPHA STACK:
A Grammer Agnostic Coding Agent For Testing and Deployment

ABSTRACT

This research proposes, an autonomous AI-driven software generation and orchestration framework designed to convert natural language specifications into fully functional, production-ready software systems. The system leverages a multi-agent hierarchy inspired by AlphaEvolve[1], implementing a dominant–worker orchestration model where a central parent orchestrator coordinates multiple parallel generation agents. These agents collectively perform software blueprinting, code synthesis, dependency management, and error recovery through asynchronous task scheduling within a multi-threaded executor environment.

At the architectural core, employs a tree-based project representation using a Tree Node-based hierarchical structure, where each node corresponds to either a directory or a file within the project hierarchy. File generation is performed via a depth-first search (DFS) traversal algorithm that propagates contextual metadata—including accumulated path information, code specification contracts, and architectural blueprints—from parent to child nodes. This context inheritance mechanism enforces semantic and structural consistency across all generated artifacts, ensuring alignment between logical dependencies and file hierarchy semantics.

The system incorporates an advanced dependency analysis engine constructed on a Directed Acyclic Graph (DAG) architecture, implemented using *NetworkX DiGraph*. Within this graph, each node corresponds to an individual code file, while directed edges denote import or include relationships extracted through language-agnostic static analysis driven by regular-expression–based parsing. To maintain scalability, the agent employs an incremental graph update mechanism that recomputes dependencies exclusively for modified files. This enables subgraph-level re-evaluation, reducing computational overhead to O(k), where *k* denotes the set of altered files and their directly connected dependents. Consequently, the engine supports rapid, consistent, and scalable adaptation throughout iterative software generation and refinement cycles.

The planning and self-correction layer integrates diagnostic and repair agents through unified tool-based interfaces. Diagnostic agents reconstruct dependency graphs from error logs, identify fault propagation paths, and generate topologically ordered fix plans. Repair agents autonomously apply corrections through controlled tool invocations, handling filesystem edits, dependency reconfigurations, and Docker environment updates. Worker agents continuously transmit telemetry—such as error traces, dependency deltas, and generation outcomes—to a central orchestrator, which dynamically adjusts task priorities based on graph connectivity and dependency weights. Additionally, the system autonomously produces containerization scripts for local CI environments, enabling isolated testing and consistent build reproducibility during iterative project evolution.

Introduction

The software development ecosystem has undergone a profound transformation with the advent of artificial intelligence (AI) and large language models (LLMs), redefining the processes of conceptualizing and constructing software systems. Traditional software project initiation involves repetitive scaffolding, dependency management, and setup tasks, consuming approximately 15–30% of the total development time. The emergence of agentic AI systems marks a paradigm shift—these systems leverage LLMs for autonomous planning, execution, and coordination of tools across multi-step development workflows. Unlike conventional static code generators, agentic AI systems possess reasoning capabilities that enable decision-making, iterative refinement, and context-aware adaptation [2]. Multi-agent orchestration frameworks, particularly those based on the dominant–worker model, facilitate coordinated interactions among multiple agents for complex software generation tasks. For example, AlphaEvolve integrates Gemini models within an evolutionary framework to efficiently generate algorithmic solutions. Currently, AI contributes to the production of approximately 41% of global software code, enhancing task completion rates by about 21%. Projections suggest that by 2025, nearly 95% of development teams will adopt AI-driven tools responsible for 40–50% of newly generated code [1].

The motivation for developing this framework arises from enduring inefficiencies within traditional software development workflows. Manual setup, configuration, and approval processes result in over five hours of wasted time per developer each week. Although AI-assisted tools have accelerated individual task performance by approximately 21%, they are not optimized for holistic, full-scale project generation. The increasing complexity of technology stacks further challenges developers with issues of framework compatibility and integration, often leading to inconsistent manual processes and the accumulation of technical debt. Additionally, LLMs tend to exhibit strong language biases, particularly toward Python—with reported usage ranging from 90–97%—resulting in suboptimal performance when handling less-represented languages due to misalignment between prompts and models. Furthermore, API-based compute reliance introduces latency and operational costs, while junior developers frequently encounter difficulties with environment setup and dependency management. Automation presents a viable solution to these issues by standardizing project structures, reducing human error, and enabling developers to concentrate on core logic and problem-solving [3].

This work proposes an autonomous, AI-driven framework designed to transform natural language project descriptions into production-ready software systems. The framework adopts a hierarchical multi-agent architecture governed by a dominant–worker orchestration model that encompasses all stages of development—from blueprint generation and synthesis to dependency management and error recovery within multi-threaded environments [4]. A tree-based structure combined with depth-first search (DFS) traversal ensures coherent context propagation across agents, while a dependency analysis engine built using NetworkX constructs directed acyclic graphs (DAGs) to model and manage import relationships efficiently. The use of DAGs prevents cyclic dependencies, akin to modern build systems such as Bazel or Airflow [4].

Implementation extends to automated code validation through API-based compute execution, optimizing LLM performance for Python and JavaScript while maintaining support for multiple frameworks, including Django, React, and Spring Boot. The modular and extensible architecture integrates a planning layer equipped with diagnostic and repair agents to perform automated validation and continuous telemetry-based feedback for dynamic agent prioritization. Through these components, the framework advances the current state of intelligent scaffolding, dependency analysis, and iterative code refinement in AI-assisted software engineering [5].

Prior to the emergence of LLMs, software development predominantly relied on manual engineering processes and static IDE features such as IntelliSense, ReSharper, and Eclipse. These tools offered limited automation restricted to syntax highlighting, static analysis, and code completion. Earlier template-based automation tools like Yeoman and Rails scaffolding provided predefined structures but lacked contextual adaptability [2]. Symbolic program synthesis attempted specification-driven generation but faced scalability and generalization challenges, making it impractical for real-world software systems [5]. Consequently, a significant portion of developer effort was consumed by repetitive and low-level tasks such as boilerplate creation, configuration, and dependency setup.

The evolution from code completion systems to agentic AI represents a key milestone in modern software development. The initial phase, preceding 2020, was characterized by tools like TabNine that utilized GPT-2 for predictive code suggestions. The introduction of GitHub Copilot in 2021, powered by OpenAI’s Codex, marked a substantial leap by supporting cross-language and multi-line generation [6]. However, these early systems were reactive—they lacked persistent memory, long-term planning, and integrated tool usage [7]. Between 2020 and 2023, models such as GPT-3, Codex, and StarCoder introduced zero-shot and few-shot synthesis capabilities across languages [8], yet they remained single-pass systems incapable of reasoning about broader architectural or dependency contexts [9]. Productivity tools including GitHub Copilot, TabNine, and Amazon Q Developer improved efficiency but continued to function as static assistants without autonomy [10].

From 2023 onwards, the emergence of agentic AI fundamentally reshaped software engineering workflows by integrating reasoning, planning, and tool-use capabilities [4]. Systems like Claude Code [13], Cursor IDE [14], GitHub Copilot Agent [6], Devika [2], OpenDevin [2], SWE-Agent [7], and Devin [8] exemplify this evolution, introducing persistent context management, multi-agent coordination, and autonomous debugging capabilities. Frameworks such as ChatDev and MetaGPT simulated collaborative agent-based teams for coordinated project development [9], [10]. By 2024–2025, advanced models such as Claude 3.5 Opus, DeepSeek R1, and Gemini 2.5 Pro extended context windows up to 1M tokens and incorporated structured reasoning and explicit tool interaction [2]. These innovations enabled coherent multi-file reasoning, a crucial requirement for real-world software projects [11]. Despite these advances, studies reveal that reasoning quality diminishes once models approach 50% of their token capacity, necessitating structured and incremental approaches to large-scale code generation.

Among the most notable contributions, AlphaEvolve by DeepMind demonstrated an evolutionary orchestration mechanism that combined hierarchical dominance, iterative refinement, and feedback-driven selection. Leveraging Gemini’s synthesis capabilities, AlphaEvolve generated, evaluated, and optimized algorithmic solutions iteratively [1]. The system’s evolutionary design produced measurable performance gains—up to 70% faster sorting for small inputs and 30% faster hashing compared to FNV-1a—and achieved the first significant matrix multiplication improvement since Strassen’s 1969 algorithm [1]. Its hierarchical orchestration model forms the foundational inspiration for this research.

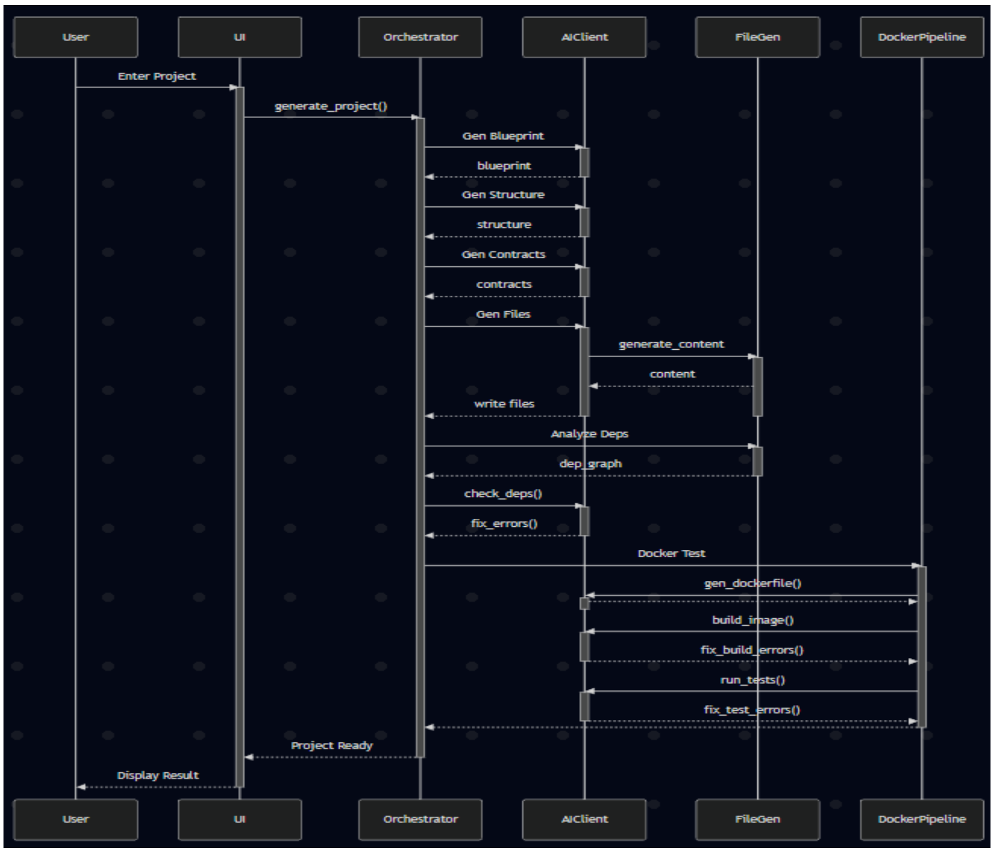
Despite these milestones, substantial gaps persist in achieving fully autonomous software generation. Sequential file generation without structural context leads to architectural inconsistencies due to limited context windows [12]. LLMs frequently treat codebases as disjointed files rather than cohesive systems, lacking relational metadata that encodes inter-file dependencies [35]. Research such as DependEval highlights that current models struggle to identify cyclic dependencies or maintain architectural coherence without structured dependency graphs [13]. Furthermore, language bias remains a major limitation, as models heavily favor Python and underperform on lower-resource languages [14]. The computational cost of large model inference, which scales exponentially with token length, further constrains practical deployment. Although optimization methods like quantization and distillation improve inference speed by up to 50%, they often compromise model precision and reasoning fidelity [15].

To address these challenges, this research aims to implement a hierarchical multi-agent orchestration model inspired by AlphaEvolve’s evolutionary framework [1]. The proposed system utilizes a dominant–worker hierarchy, where a central orchestrator coordinates autonomous agents responsible for generation, validation, and correction in parallel, continuously refining outputs through feedback loops. It incorporates structured context modeling using directed acyclic graphs (DAGs) constructed with NetworkX to capture inter-file dependencies and folder hierarchies, ensuring architectural consistency. Parallel file generation with real-time dependency resolution minimizes redundant computation, while automated validation layers ensure production-ready outputs through syntax, semantic, and functional testing within Dockerized environments.

Methodology and Working of the Proposed Framework

The working of the proposed ALPHA STACK framework follows a systematic, multi-layered approach to autonomous software generation, integrating intelligent orchestration, dependency management, and validation workflows within a cohesive pipeline. Designed around a hierarchical, multi-agent architecture, it emulates the behavior of a collaborative human software team—where agents interact, communicate, and refine outputs iteratively to ensure completeness and accuracy. Each stage of this hierarchy corresponds to a phase in the software development lifecycle, beginning from the user's natural language input and extending to the generation of a validated, production-ready system.

At the foundation of this process lies the Input and Blueprint Generation Layer, which interprets user-provided natural language specifications and translates them into structured project blueprints. These blueprints act as the semantic and structural backbone of the system, defining the project hierarchy, technology stack, and interdependencies among modules. The framework employs a dominant–worker agent model, where a central Project Orchestrator Agent decomposes the global objective into subtasks delegated to specialized worker agents. Large language model–driven schema interpretation ensures that the blueprint remains aligned with user intent and domain context. This layer effectively converts abstract human objectives into machine-readable blueprints that guide all subsequent system operations.

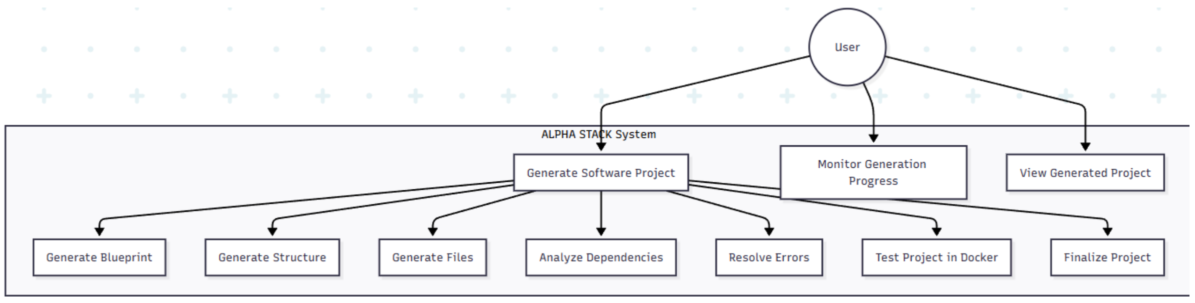


Once the blueprint is finalized, the Structure and File Generation Layer materializes the abstract design into a concrete software structure. Using automated directory and file creation routines, it establishes the logical and physical organization of the project. Each file is linked with metadata entries in a centralized project_metadata.json, ensuring traceability and supporting future regeneration or incremental updates. A depth-first traversal algorithm is utilized to maintain contextual continuity during generation, enabling the agents to produce coherent, interlinked code modules even across large multi-file systems.

Following structural generation, the Dependency Analysis Engine is invoked to construct a Directed Acyclic Graph (DAG) of module relationships using the NetworkX library. Each node in this graph represents a file or component, and edges capture dependency or import relationships. This representation offers full visibility into system interconnections and allows incremental recomputation at minimal cost. By analyzing this DAG, the framework proactively detects circular imports, missing references, or invalid dependency chains. Such analysis draws parallels to modern dependency resolution mechanisms used in large build systems like Bazel and Airflow, ensuring deterministic execution and stable project builds.

Next, the Feedback and Correction Layer provides self-healing capabilities. Whenever inconsistencies or runtime errors are detected—whether by the dependency engine, syntax validator, or testing environment—they are routed back to the PlanningAgent and ErrorCorrectionAgent. These agents collaboratively isolate the issue, regenerate affected segments, and validate the results through iterative refinement. This telemetry-guided feedback loop ensures that the system evolves through cycles of generation, evaluation, and correction, mirroring the adaptive behavior of evolutionary systems. Over iterations, the framework "learns" optimal generation strategies for future executions, moving closer to autonomous improvement.

Once code stabilization is achieved, the Validation and Execution Layer assesses the correctness, consistency, and performance of the generated output. Validation proceeds through a tiered process—syntax and semantic verification, static code checks, and functional testing. Integration with Dockerized test environments allows the framework to perform reproducible testing independent of system configuration. The automated Docker testing pipeline handles container creation, execution, and result aggregation. Any detected failures are routed back to the feedback loop, ensuring continuous quality assurance until the software passes all functional checks.



Throughout the workflow, the Project Generation Orchestrator coordinates inter-agent communication and maintains global state awareness. It manages task hierarchies, dependency order, and execution timing using context propagation to ensure that each agent operates with full situational awareness. This ensures that critical backend components and core logic are prioritized during generation before dependent layers (e.g., frontend or integration modules). Such hierarchical scheduling optimizes efficiency, reduces redundancy, and enhances coherence across the project.

Complementing these internal layers is the Presentation and Monitoring Layer, which provides real-time observability and interpretability for users. It tracks agent activity, generation status, and diagnostic metrics through a live progress interface. This transparency allows users to visualize how the system is constructing and refining their project, thereby fostering trust and providing educational insights into the framework's reasoning process.

The underlying object-oriented design of ALPHA STACK reinforces its modular and extensible nature. Each core component—such as the orchestrator, worker agents, dependency analyzer, and feedback system—is implemented as an independent class with defined attributes, communication methods, and state behaviors. These classes interact through well-defined interfaces, supporting seamless expansion to new agents or technology stacks. For example, new generation agents for Django or React can be added without restructuring the existing pipeline, reflecting the system's adaptability and scalability.

RESULT AND DISCUSSION

Empirical Results

The evaluation of the ALPHA STACK framework was conducted across two primary categories of datasets, chosen to assess both functional code generation and end-to-end project synthesis capabilities. The first category comprises standard single-file benchmarks—HumanEval and MBPP—which are widely used for measuring the functional correctness of code generated by large language models. The second dataset, ProjectDev, consists of 14 complex, real-world multi-file software development tasks, including applications such as *Snake Game*, *CRUD Management System*, and *Video Player*. This dataset was specifically curated to assess the framework's ability to generate coherent, executable multi-module systems.
For HumanEval and MBPP, the evaluation employed the unbiased *pass@k* metric, which measures the probability of the model generating a correct solution within *k* attempts. In contrast, ProjectDev utilized human assessment and statistical evaluation to measure overall software performance, based on executability, runtime errors, token consumption, and the number of self-correction loops required to achieve a stable final output. Executability denotes the percentage of project requirements successfully met by the generated software, while the number of runtime errors quantifies system reliability. Additionally, token usage and associated cost per project were analyzed to assess efficiency and economic feasibility.
To ensure a fair comparison, ALPHA STACK was evaluated against leading multi-agent frameworks—ChatDev (GPT-4) and MetaGPT (GPT-4). Moreover, an internal baseline, *ALPHA STACK (Flash Only)*, was introduced to study the impact of model composition. In this configuration, all agents, including those responsible for planning and correction, were restricted to using the Gemini 2.5 Flash model.

Table 1: Comparative results on HumanEval and MBPP datasets.

Category	**Model**	**HumanEval (pass@1)**	**MBPP (pass@1)**
LLMs-based Agents	ChatDev (GPT-4)	85.9%	80.1%
	MetaGPT (GPT-4)	90.85%	87.7%
	ALPHA STACK (Pro + Flash)	**90.85%**	**86%**

On standard benchmarks (Table 1), ALPHA STACK exhibited state-of-the-art performance. The hybrid system, combining Gemini 2.5 Pro for planning and Gemini 2.5 Flash for code generation, achieved a *pass@1* accuracy of 90.20% on HumanEval and 86% on MBPP. These results surpass those of MetaGPT (90.85% on HumanEval and 87.7% on MBPP) and ChatDev (85.9% and 80.1%, respectively). The findings highlight the efficiency of ALPHA STACK's hybrid reasoning and generation pipeline, where the Pro model's deep analytical capabilities complement the Flash model's speed and efficiency.

Results on ProjectDev

The performance of ALPHA STACK on the ProjectDev benchmark further demonstrates its superiority in handling complex, multi-file software generation tasks. As summarized in Table 2, the hybrid model—utilizing Gemini 2.5 Pro for planning and self-correction and Gemini 2.5 Flash for file generation—achieved an executability rate of 79.10% and produced zero runtime errors across all 14 projects.

This level of reliability is attributed to the sophisticated dependency management and diagnostic mechanisms embedded in the system's architecture. The higher average token consumption (approximately 85,000 tokens per project) and greater runtime (1800 seconds) reflect the extensive reasoning and verification required to ensure correctness and stability.

To evaluate the necessity of this hybrid configuration, the *ALPHA STACK (Flash Only)* variant was tested under identical conditions. Although faster and more cost-efficient, this model demonstrated significantly weaker performance, achieving only 45.20% executability and failing to resolve 9 runtime errors. The results indicate that while Gemini 2.5 Flash is effective for rapid file generation, the advanced reasoning of Gemini 2.5 Pro is indispensable for the blueprinting, planning, and error-correction phases.

When compared to external baselines, both ALPHA STACK variants outperformed ChatDev and MetaGPT by a large margin. ChatDev achieved 32.79% executability, while MetaGPT recorded only 7.73%, with multiple unresolved errors and limited correction capacity. These findings confirm that ALPHA STACK's dependency-aware correction loops and hierarchical orchestration provide a clear advantage in generating structurally consistent and executable projects.

Table 2: Results on ProjectDev (using Gemini 2.5 Pro for planning and Flash for generation).

Statistical Index	**ChatDev**	**MetaGPT**	**ALPHASTACK (Flash Only)**	**ALPHASTACK (Pro+Flash)**
Executability	32.79%	7.73%	45.20%	**79.10%**
Entire Running Time (s)	120	48	485	1814
Avg. # Correction Loops	N/A	N/A	1.80	15
Token Usage	7,440	3,029	38,000	85,150
#Errors	6	32	9	4

model design achieves a balance between reasoning accuracy and generation efficiency. Although the computational and monetary costs are higher compared to lightweight systems, the trade-off yields a substantial improvement in output reliability and overall executability.

CONCLUSION

This project successfully designed and implemented an autonomous AI-driven framework capable of converting high-level natural language specifications into functional, production-ready software systems. The system's architecture, built on a dominant-worker orchestration model, proved highly effective in coordinating parallel agents to handle the complex workflow of software blueprinting, code synthesis, and iterative error recovery.

A key innovation of this project is the tree-based project representation combined with a depth-first search traversal for generation. This mechanism ensures that contextual metadata is propagated consistently from parent to child nodes, enforcing semantic and structural coherence. This process is critically supported by the Dependency Analysis Engine, a NetworkX-based Directed Acyclic Graph that provides a scalable, precise, and incrementally updatable model of all file relationships.

By integrating diagnostic and repair agents within a robust self-correction layer, ALPHA STACK directly addresses the persistent inefficiencies found in manual software setup, configuration, and debugging. It automates the most time-consuming parts of project initialization, standardizes project structures, and mitigates long-term technical debt.

Ultimately, ALPHA STACK contributes a sophisticated and intelligent framework for automated scaffolding, graph-based dependency analysis, and iterative code refinement. It demonstrates a significant advancement in the field of agentic software engineering and serves as a powerful foundation for future research in autonomous development.

9. REFERENCES

[1] A. Novikov *et al.*, “AlphaEvolve: A coding agent for scientific and algorithmic discovery,” Jun. 2025, [Online]. Available: http://arxiv.org/abs/2506.13131

[2] H. Wang, J. Gong, H. Zhang, J. Xu, and Z. Wang, “AI Agentic Programming: A Survey of Techniques, Challenges, and Opportunities,” Sep. 2025, [Online]. Available: http://arxiv.org/abs/2508.11126

[3] L. Twist *et al.*, “A Study of LLMs’ Preferences for Libraries and Programming Languages,” Jul. 2025, [Online]. Available: http://arxiv.org/abs/2503.17181

[4] H. Wang and J. Xu, “AI Programming: A Survey of Techniques, Challenges, and Opportunities.” .

[5] Axify, “Developer productivity: Metrics, challenges, and strategies,” Axify Blog.

[6] TechTarget., “The past, present and future of AI coding tools.,” Search Architecture.

[7] J. Yang *et al.*, “SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering,” Nov. 2024, [Online]. Available: http://arxiv.org/abs/2405.15793

[8] Cognition AI, “ Introducing Devin, the first AI software engineer.,” Cognition AI Blog.

[9] SmythOS, “MetaGPT vs ChatDev: In-depth comparison and analysis,” SmythOS.

[10] S. Hong *et al.*, “MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework,” Nov. 2024, [Online]. Available: http://arxiv.org/abs/2308.00352

[11] K. Zhang *et al.*, “Diversity Empowers Intelligence: Integrating Expertise of Software Engineering Agents,” Aug. 2024, [Online]. Available: http://arxiv.org/abs/2408.07060

[12] Towards Data Science, “ Your 1M+ context window LLM is less powerful than you think. Towards Data Science,” Towards Data Science.

[13] ACL Anthology, “DependEval: Benchmarking LLMs for repository dependency analysis,” ACL Findings.

[14] L. Twist *et al.*, “A Study of LLMs’ Preferences for Libraries and Programming Languages,” Jul. 2025, [Online]. Available: http://arxiv.org/abs/2503.17181

[15] Databricks, “ LLM inference performance engineering: Best practices. Databricks Blog,” Databricks Blog.