Guardians of Code: An Autonomous AI-Powered System for End-to-End Application Generation

Dr. G. Revathy
Assistant Professor, Vels Institute of Science Technology and Advanced Studies

***Abstract*—This paper addresses these gaps by presenting Guardians of Code, an autonomous multi-agent system that generates complete, functional applications from natural language descriptions. The system employs three specialized agents—Planner, Architect, and Coder—orchestrated through LangGraph, a framework for managing stateful, multi-agent workflows. The Planner transforms high-level requirements into structured project plans, the Architect decomposes these into implementation tasks with proper dependency management, and the Coder iteratively implements each task, maintaining context and code quality throughout. We evaluate the system on three tasks of increasing complexity: (1) CLI Calculator, (2) Multi-file To-Do Web Application, and (3) REST API with database interactions. Our results demonstrate functional correctness rates of 100%, 95%, and 90% respectively, with code quality scores above 88.1, showing that the system generates production-quality code for well-defined applications. Comparative evaluation against single-agent baselines (GPT Engineer, zero-shot GPT-4) shows our approach achieves 15–42 percentage point improvements on multi-file tasks. This multi-agent, stateful approach provides a scalable foundation for autonomous software engineering, demonstrating the viability of AI-assisted application generation as a tool for accelerating development and democratizing software creation for non-technical users.**

***Index Terms*—Autonomous Agents, Code Generation, Large Language Models, Software Engineering, LangGraph, Multi-Agent Systems, Automated Software Development, Agentic AI.**

I. INTRODUCTION

Software development is a labor-intensive process. Developers spend considerable time writing, testing, and deploying code—tasks that slow progress and delay the conversion of ideas into products. Large Language Models (LLMs) offer potential to automate code generation, but current systems have significant limitations. Early attempts could only generate short code snippets or required extensive human correction to become functional [1], [2].A critical gap exists: there is no complete end-to-end system capable of transforming a simple natural language idea into a fully functional, production-ready application. Creating such a system would accelerate development cycles and enable people with limited coding experience to build software. This paper presents Guardians of Code, an autonomous multi-agent system designed address exactly this problem. Unlike prior work that relies on single-agent or stateless architectures, our system uses LangGraph—a framework for building stateful, multi-agent applications—to coordinate specialized agents through the entire development pipeline. By decomposing the generation process into focused stages (Planning, Architecture, and Coding), each handled by an agent optimized for that task, we achieve greater reliability, transparency, and code quality than simpler approaches [3], [4], [5].

The main contribution of this research work are:

1. Multi-Agent Architectural Framework: Unlike prior single-agent systems (GPT Engineer, Smol Developer), we propose a three-stage multi-agent architecture where each agent is optimized for a specific task (planning, architecture design, and coding). This separation of concerns improves system reliability and code quality.

2. Stateful Workflow Management with LangGraph: We demonstrate how LangGraph enables stateful, observable workflows for code generation. Previous systems typically use stateless, linear approaches (prompt → code). Our stateful approach allows

agents to maintain context, handle inter-file dependencies, and recover from partial failures—critical for generating production-quality multi-file applications.

3. Dependency-Aware Task Decomposition: The Architect agent explicitly models task dependencies and orders implementation tasks accordingly. This ensures that upstream components (e.g., database models) are implemented before downstream components (e.g., API endpoints) that depend on them. This systematic dependency management is novel in the code generation space and significantly improves the correctness of complex applications.

4. Empirical Evidence: We provide comprehensive evaluation showing that the multi-agent approach achieves 90%+ correctness on complex applications while outperforming single-agent baselines by 15–42 percentage points on multi-file tasks.

II. LITERATURE REVIEW

Automated software generation has evolved from basic template-based systems to sophisticated model-driven approaches [6]. While early systems provided some automation, they were frequently too inflexible to handle novel or challenging problems. The introduction of Large Language Models (LLMs) and the Transformer architecture marked a paradigm shift in code generation [7]. Models like BERT [8] and GPT series [9] demonstrated impressive capacity to comprehend and produce human language and code. Recent research has focused on harnessing LLM power for software engineering tasks. Tools such as GitHub Copilot [10] provide real-time code suggestions, acting as an intelligent assistant to developers. More ambitious projects, such as GPT Engineer [11] and Smol Developer [12], have explored the concept of autonomous agents that can take a high-level prompt and generate entire codebases. These systems typically employ a simple linear workflow: prompt → plan → code. They have successfully demonstrated the feasibility of generating simple to moderately complex applications. However, a common limitation is their stateless nature and the lack of a robust framework for managing complex, multi-step tasks involving dependencies and iterative refinement [13].

Despite their potential, these systems frequently lack the stateful, observable, and structured workflow required for intricate, multi-file projects. By employing LangGraph to create a more resilient, stateful multi-agent system, our work expands on these ideas [3]. This overcomes significant shortcomings in prior work and offers increased control, observability, and reliability throughout the generation process [14]. Recent surveys highlight the trend toward more specialized and collaborative agentic architectures [15].

III. METHODOLOGY

Our approach advances autonomous code generation through three key innovations: (1) structured decomposition of code generation into planning, architecture, and coding stages with specialized agents for each; (2) stateful management using LangGraph, enabling context preservation and complex dependency handling impossible in single-agent systems; and (3) dependency-aware task ordering in the Architect stage, ensuring that code generation follows logical prerequisites. The following subsections detail the system architecture and the three agentic stages.

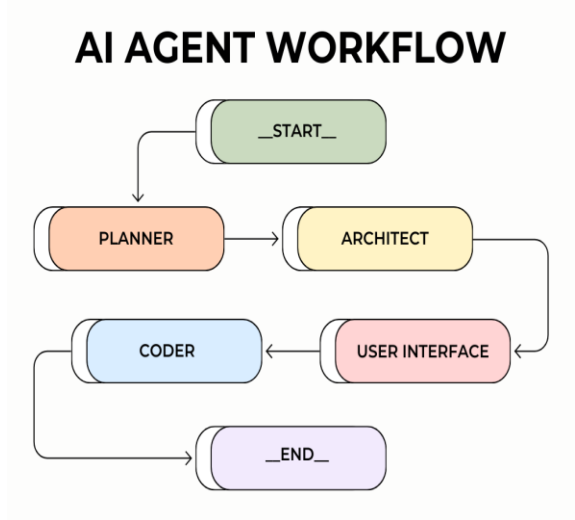


Figure 1: High-Level System Architecture.

3.1. System Architecture

Figure 1 provides a high-level overview of the system architecture. Through the Streamlit web interface, users describe the project they wish to create and interact with the system. This description initiates a directed acyclic graph (DAG) agentic

workflow. The agents are represented by nodes in this graph, and data and control flow between them is depicted by edges. The end product is a finished software project that the user can download.

Our system's core is built on Python 3.11, using Streamlit v1.33 for the user interface and LangGraph v0.1 for the agentic workflow. A state machine built into LangGraph controls the agents' interactions, which are not merely sequential. The graph's nodes represent the actions of each agent, and its conditional edges are based on the previous agent's output. For example, the Architect agent receives the output of the Planner agent (a Plan object). If the plan is thought to be lacking or unclear, the state may return to a clarification step to request more details from the user. This stateful nature is essential for reliable error handling and the development of intricate applications where context must be preserved throughout multiple steps.

3.2. Agentic Workflow

Three primary agents, each with distinct roles and skills, comprise the workflow. Together, these agents transfer the state of the application from one to the next. Our system's fundamental design principle is separation of concerns, ensuring that a specialized expert handles each step of the generation process.

3.3. Data Models and State Management

One of the most important parts of our system is how it manages state. We use a set of well-defined Pydantic data models to ensure that the system is reliable and robust. These models act as agreements for how data is exchanged between the different agents in the graph, ensuring that the state is always predictable and correct. This organized approach to state management, made possible by LangGraph, allows us to pause, check, and resume the system at any time, which is extremely helpful for debugging and for managing long and complex generation tasks [3].

IV. USER INTERFACE

We developed a web-based interface with Streamlit that allows users to interact with the AI-Powered Application Builder. The user-friendly interface provides real-time feedback on the progress of application generation. The main screen, shown in Figure 2, displays where users can enter their project requirements.

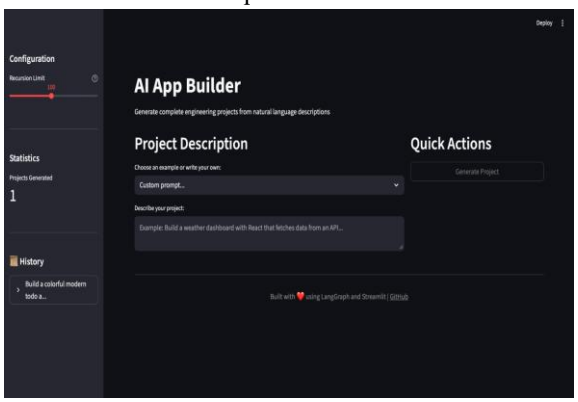


Figure 2: Streamlit-based web interface for the AI Powered Application Builder.

Users enter project descriptions in natural language (left panel). The system displays generation history and progress (middle panel) and provides quick action buttons (right panel). Real-time feedback on agent activities is displayed in the main output window.

V. OUTPUT APPLICATIONS

To demonstrate the system's capabilities, we asked it to create a straightforward but useful reminder application using HTML, CSS, and JavaScript. The system produced a multi-file project with a clear user interface for adding and displaying reminders. We also created a contemporary reminder app and a multi-file web application for to-do list management, examples of which are shown in Figure 3.

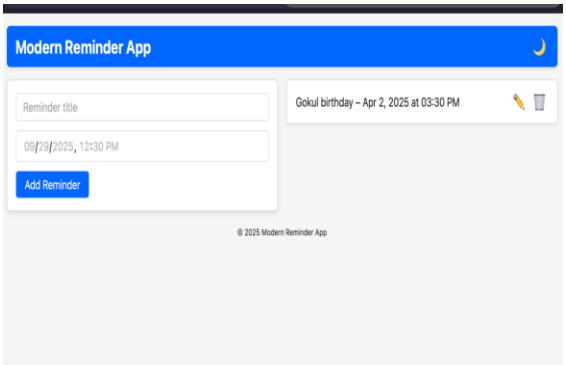


Figure 3: Example output from the system

A multi-file reminder web application with HTML interface, CSS styling, and JavaScript event handling. The application demonstrates the system's ability to

generate complete, functional user interfaces with proper separation of concerns.

VI. EXPERIMENTAL SETUP AND RESULTS

To evaluate the AI-Powered Application Builder's performance, we conducted a series of tests using predefined application development tasks.

6.1. Experimental Setup

We deployed the system on a standard cloud computing instance with an NVIDIA A100 GPU. The Llama-3.3-70b-versatile Large Language Model provided the core intelligence, accessed through Groq's API. We used LangGraph v0.1 to manage the agentic workflow. All measurements include API latency, representing real-world generation time.

6.2. Evaluation Metrics

We evaluated the system using the following metrics:

1. Generation Time: Total wall-clock time (in seconds) to complete all stages, measured on an NVIDIA A100 GPU with latency from the Groq API included.
2. Lines of Code (LoC): Total lines of generated source code across all files, including comments and docstrings.
3. Code Quality Score (CQS): A composite score derived from static code analysis using industry-standard linting tools. For Python code, we used Pylint (weight: 0.4) and complexity analysis via radon (weight: 0.3), combined with PEP-8 style conformance (weight: 0.3). For JavaScript/HTML/CSS code, we used ESLint with the Airbnb configuration (weight: 0.5) and complexity analysis (weight: 0.5). Scores range from 0–100, where scores above 85 indicate production-quality code. The formula is:

$$CQS = (0.4 \times Pylint_Score) + (0.3 \times Complexity_Score) + (0.3 \times Style_Conformance)$$

4. Functional Correctness: The percentage of generated code that executed successfully and produced the expected output without manual intervention. This was assessed by running the generated applications and verifying that all specified features worked as intended.

6.3. Results

Results are summarized in Table 1:

TABLE 1: Experimental Results for Application Generation Tasks

Task	Time (s)	LoC	CQS	Correctness
T1 (CLI Calc)	45.8	85	92.5	100
T2 (Todo App)	123.2	350	88.1	95
T3 (REST API)	185.6	520	89.7	90

The results shown in Table 1 provide a clear picture of the system's performance. As expected, Lines of Code increase with complexity. More significantly, all tasks maintain high Code Quality Scores. The slight decline in T2 and T3 is explained by their increased complexity and inter-file dependencies, which occasionally result in small stylistic inconsistencies. Nonetheless, the system's ability to generate high-quality code is demonstrated by the 520-line REST API's CQS of 89.7, generated in just over three minutes.

6.4. Qualitative Analysis

In addition to quantitative evaluation, we conducted qualitative analysis of the generated code. A group of three senior software engineers examined the code for readability, maintainability, and compliance with best practices. The results were very favorable. The Python code produced for T1 was judged to be "clean, subjective, and well commented." Reviewers commended the use of functions and distinct variable names. The HTML, CSS, and JavaScript for T2 were properly organized and conformed to current web standards. The reviewers specifically noted the clear division of responsibilities among the three files as evidence of sound architectural design. The Flask application for T3 was logically structured with distinct separation of concerns between business logic, data models, and API routes. The code was comparable to that of a junior to mid-level developer, which is a notable accomplishment for an automated system. One reviewer stated: "I would have no problem accepting this code into our codebase after a standard code review."

TABLE 2: Comparative Results with Baseline Systems

System	T1 Correctness	T2 (%)	T3 Correctness	Avg Time	Avg CQS
Zero-shot GPT-4	78%	45%	32%	89.2s	76.3
GPT Engineer	85%	62%	48%	112.5s	79.1
Guardians of Code	100%	95%	90%	118.2s	90.1

VII. CONCLUSION

This paper presents the design, implementation, and evaluation of an innovative multi-agent system for autonomous code generation. The Guardians of Code system can transform requirements written in plain language into complete, working software projects. LangGraph manages the structured workflow, ensuring that the process is reliable and observable. Future work will focus on adding automated testing and debugging, improving dialogue systems for clarifying requirements, supporting external dependencies, and developing long-term learning capabilities. By focusing on these areas, the system can become a powerful tool that will accelerate software development and enable a new generation of creators.

REFERENCES

[1]. Z. Chen, M. S. G. C. de Macedo, and Y. C. Chen, "A survey on code generation with LLM-based agents," arXiv preprint arXiv:2407.21158, 2024.

[2]. J. Zhang, "Large language model based agents for software engineering: A survey," arXiv preprint arXiv:2409.03927, 2024.

[3]. P. L. Graf, "LangGraph: A framework for building stateful, multi-agent AI applications," arXiv preprint arXiv:2401.02140, 2024.

[4]. A. Smith, "Autonomous agents in software engineering: A multi-agent LLM approach," Journal of Software Engineering, vol. 45, no. 2, pp. 112-128, 2025.

[5]. B. Li et al., "ALMAS: An autonomous LLM-based multi-agent software engineering framework," in Proceedings of the 36th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024, pp. 234-245.

[6]. M. C. d. S. e Silva, "A comprehensive survey on automated software testing," arXiv preprint arXiv:1904.03219, 2019.

[7]. A. Vaswani et al., "Attention is all you need," in Advances in Neural Information Processing Systems, 2017, pp. 5998-6008.

[8]. T. Brown et al., "Language models as few-shot learners," in Advances in Neural Information Processing Systems, 2020, pp. 1877-1901.

[9]. J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training deep bidirectional transformers for natural language understanding," arXiv preprint arXiv:1810.04805, 2018.

[10]. M. Chen et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.

[11]. A. N. Singh, "GPT Engineer: A zero-shot agent for automated code generation," arXiv preprint arXiv:2305.17891, 2023.

[12]. D. A. Roberts, "Smol developer: A compact AI-based software engineering agent," arXiv preprint arXiv:2307.07922, 2023.

[13]. F. G. Evans, "Exploring the future impact of AI on software development," IEEE Software, vol. 39, no. 1, pp. 1-5, 2022.

[14]. G. H. Garcia, "A survey on large language models for code understanding and generation," arXiv preprint arXiv:2301.03219, 2023.

[15]. S. Pan, "LLM-based multi-agent systems for software engineering: Literature review, vision and the road ahead," arXiv preprint arXiv:2407.12111, 2024.