OrchestraAI: A Multi-Agent Generative AI for Software Development

João Luiz Bione da Silva Holanda, Thiago Silva de Souza

Centro Universitário Lasalle do Rio de Janeiro (Unilasalle-RJ)
Niterói – RJ – Brazil

joao.bione@icloud.com, profthiagodesouza@gmail.com

Abstract. *This paper presents OrchestraAI, a multi-agent AI assistant designed to support professional software developers by combining conversational interaction with autonomous execution of development tasks, including code generation and Git version control. Unlike typical AI chatbots, OrchestraAI actively performs actions such as creating and modifying code files within a customizable local environment, preserving privacy and offering flexibility through the integration of open-source and external language models. The paper details the system's modular architecture, which leverages LangChain and pluggable Large Language Models (LLMs), and reports qualitative results from initial evaluations. The findings indicate a strong potential for productivity improvement, along with identified limitations and proposed future improvements.*

1. Introduction

Recent advances in artificial intelligence (AI) have significantly impacted software development, particularly through the emergence of intelligent coding assistants. Tools such as GitHub Copilot and OpenAI ChatGPT can generate code snippets or entire functions from natural language prompts, facilitating developer productivity. However, these solutions typically rely on proprietary large language models (LLMs) and require paid subscriptions or continuous cloud connectivity, which may not align with the needs and constraints of all developers. Consequently, there is a growing demand for AI-assisted development approaches that are cost-flexible, privacy-preserving, and customizable, allowing developers to choose or switch between underlying LLMs according to their specific requirements and workflows. This demand is underlined by recent scholarly analyses of the local deployment of language models on devices, which emphasize enhanced privacy (no code leaves the device), offline capability, lower latency, and reduction of recurring costs Xu et al. [2024].

To address these needs, OrchestraAI is proposed as a local, developer-oriented AI assistant that extends beyond the limitations of conventional chatbot-based systems. Rather than restricting interactions to question-answer dialogues, OrchestraAI executes concrete development tasks, including creating and modifying code files, executing system commands, and managing version control operations, all through natural language instructions. The system is designed to operate within the developer's local environment—such as a personal machine or integrated development environment (IDE)—ensuring that source code and data remain private while enabling the use of open-source models for inference. Although developers may optionally connect OrchestraAI to external LLMs via paid APIs if desired, the core system remains independent of paid services, offering flexibility in cost and deployment.

The system was conceived and developed with the goal of advancing AI-assisted programming beyond chat-based interactions by incorporating multiple specialized agents, each responsible for specific aspects of the development workflow. This multi-agent design enables OrchestraAI to interact naturally with users while autonomously executing development tasks under user guidance. The target audience includes professional developers seeking to leverage AI to streamline workflows without sacrificing control or incurring additional costs. Functioning as a 'co-developer', OrchestraAI handles routine and multi-step operations, enabling developers to focus on higher-level problem-solving and design activities. In summary, OrchestraAI represents a novel approach to AI-assisted software development by offering an extensible, multi-agent system that operates within the user's environment, adapts to individual developer preferences, and automates a wide range of tasks, from coding to version control.

The remainder of this paper is organized as follows. Section 2 presents related work on generative AI and multi-agentic systems in software engineering. Section 3 details the architecture and implementation of OrchestraAI, describing its specialized agents and orchestration workflow. Section 4 reports and discusses the qualitative evaluation results, highlighting capabilities, limitations, and opportunities for improvement. Section 5 presents the threats to validity of this study, outlining the limitations of the current evaluation and identifying avenues for strengthening future assessments. Finally, Section 6 concludes the paper and outlines future work directions.

2. Literature Review

In recent years, *Generative AI*—particularly large-scale models such as *Large Language Models (LLMs)*—has ushered in a new era of AI-driven content creation. Artificial intelligence-generated content (AIGC) refers to media (text, images, etc.) produced by AI algorithms rather than humans, enabling the rapid generation of high-quality content in a short time frame Cao et al. [2024]. Prominent examples include *ChatGPT*, an conversational LLM by OpenAI, and *DALL-E 2*, an image generator, which together illustrate how generative models now create human-like text and novel imagery on demand Cao et al. [2024]. These advances make content creation more efficient and accessible, as AI can automate the production of substantial content in very little time.

Rapid progress in generative AI has been largely driven by scale and innovation in model training. Modern foundation models leverage vast datasets, massive neural network architectures with billions of parameters, and extensive computational resources to achieve unprecedented capabilities. For example, GPT-3 retained the transformer-based architecture of GPT-2 but grew from 1.5 billion to 175 billion parameters and trained on orders of magnitude more data (CommonCrawl vs. smaller WebText), which produced substantially improved generalization and content quality Cao et al. [2024]. Such a scaling has markedly improved the realism and coherence of the generated output.

In parallel, new techniques such as *Reinforcement Learning from Human Feedback (RLHF)* have been introduced to align the model outputs with human preferences and correctness. InstructGPT and ChatGPT apply RLHF during fine-tuning: human labelers rank model output, and these rankings are used to train a reward model, which then guides the LLM through a policy optimization step, significantly improving reliability and factual accuracy over time Ouyang et al. [2022]. These innovations help address the questions of

relevance and factuality in the generated content by iteratively refining the behavior of the model based on human-guided signals.

Beyond natural-language content, large language models have been adapted for software engineering tasks such as code generation and developer assistance. When integrated into developer environments, these models can translate natural-language descriptions into functional source code and provide context-aware completions, refactoring suggestions and optimization guidance Chen [2021]. These capabilities are changing how programmers interact with code, enabling faster production of boilerplate and simple functions while supporting tasks like documentation generation and code review Chen [2021]. Parallel research has also explored specialized applications, such as generating commit messages from code diffs using LLMs, demonstrating the potential of language models to automate routine software engineering tasks. Nevertheless, current LLMs still face challenges related to code quality and reliability, underscoring the need for human oversight and specialized agentic frameworks that can mediate between the model and the development workflow.

Beyond generation, researchers have begun designing autonomous AI *agents* that couples the reasoning and language understanding of an LLM with tools to act on the world, executing code, browsing the Web, or controlling software. These agents can operate continuously, handling repetitive or multi-step tasks without human intervention. A prime example is AutoGPT, an open source framework that autonomously decomposes high-level goals into subtasks (web searches, API calls, email drafting) and executes them without further user guidance Ning et al. [2025]. In the Web domain, the so-called *WebAgents* observes the content of the page, uses an LLM to decide each browser action, executes it, and repeats until the original command is complete Ning et al. [2025].

As agents gain autonomy and tool-use capabilities, new security and alignment challenges arise. Narajala & Narayan (2025) introduce the *Advanced Threat Framework for Autonomous AI Agents (ATFAA)* and a complementary mitigation suite (*SHIELD*) to address novel vulnerabilities: from prompt injection attacks in the agent's reasoning pipeline to poisoning of its long-term memory module Narajala and Narayan [2025]. This work highlights the need for new security paradigms tailored to agentic AI.

The traditional single-agent model has advanced into agentic AI, characterized by multiple specialized agents working collaboratively to address complex problems. In this paradigm, individual agents with distinct roles coordinate dynamically, decompose high-level tasks into actionable subtasks, maintain persistent context across interactions, and operate with a degree of autonomy while ensuring alignment with overarching goals. This multi-agent orchestration builds upon foundational concepts in multi-agent systems while leveraging the adaptability of modern generative models, enabling emergent and cooperative behaviors even in unstructured and evolving environments Sapkota et al. [2025].

Despite these advancements, key challenges persist in the deployment of generative and agentic AI systems. Agents may produce hallucinations or factual inaccuracies with high confidence, posing risks to reliability. Their performance can degrade significantly when exposed to novel or out-of-distribution inputs, reflecting brittleness in real-world scenarios. In multi-agent environments, emergent behaviors may lead to unanticipated

cooperation or conflicts, complicating control and predictability. Furthermore, the autonomy and tool-use capabilities of these agents introduce new security vulnerabilities, necessitating the development of robust safeguards for safe and effective integration into user workflows.

3. OrchestraAI

OrchestraAI is a specialized multi-agent generative AI system designed explicitly to facilitate software development through human-AI interaction loops. Drawing from contemporary advances in Generative AI and leveraging Large Language Models (LLMs), the OrchestraAI architecture orchestrates tasks related to version control, code generation, testing, and conversational interaction to enhance developer productivity.

3.1. System Architecture

The architecture of OrchestraAI consists of three specialized agents, each responsible for a distinct domain-specific function and coordinated by a central orchestrator that manages task distribution and workflow execution. This structure is represented in Figure 1.

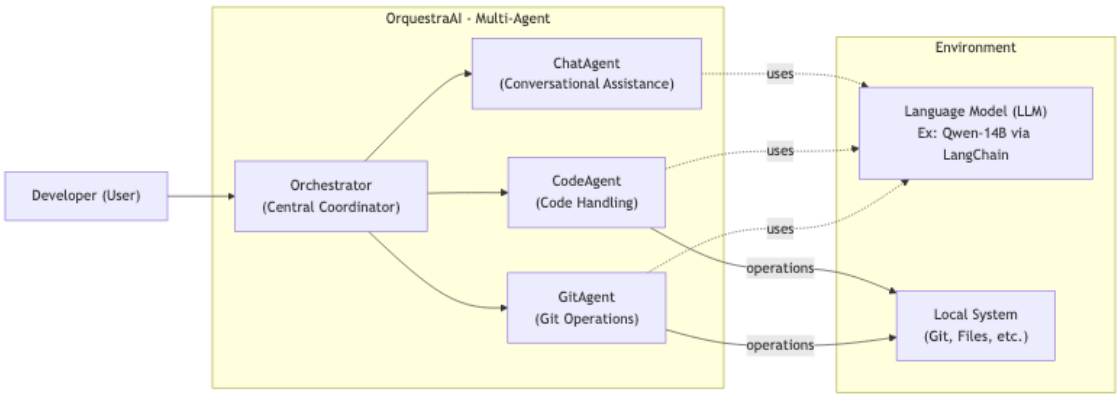

Figure 1. System Architecture

- **GitAgent**: Manage Git version control operations, generating semantic commit messages adhering to the Conventional Commits specification, handling branch management, and automating repository interactions such as status checks and diffs.
- **CodeAgent**: Performs a variety of software engineering tasks, including file creation, editing, and reading. It supports multiple programming languages and frameworks, automatically executes and generates test cases, performs static code analysis and refactoring suggestions, and manages overall project structures and dependencies.
- **ChatAgent**: Provides a natural-language-based interface for developers, answering general technical questions, helping in documentation creation, and offering explanations of programming concepts and best practices.

The central **Orchestrator** routes user requests to the appropriate agent, manages task prioritization, handles errors gracefully, and ensures smooth inter-agent communication and workflow execution.

3.2. Interaction Workflow

The interaction with OrchestraAI typically follows a structured conversational workflow, as illustrated in Figure 2. The developer begins by issuing a request using a natural language prompt (e.g., "Create a Python file with basic mathematical functions"). The orchestrator then parses this request, identifies the appropriate agent(s) to handle it, and forwards the command accordingly. The selected agent (GitAgent, CodeAgent, or ChatAgent) executes the task and produces an intermediate output, such as code snippets, test cases, or commit messages. The developer can review this output and, if necessary, provide iterative feedback, allowing the agent to refine its responses and actions, thereby exemplifying the Vibe Coding paradigm within the development workflow.

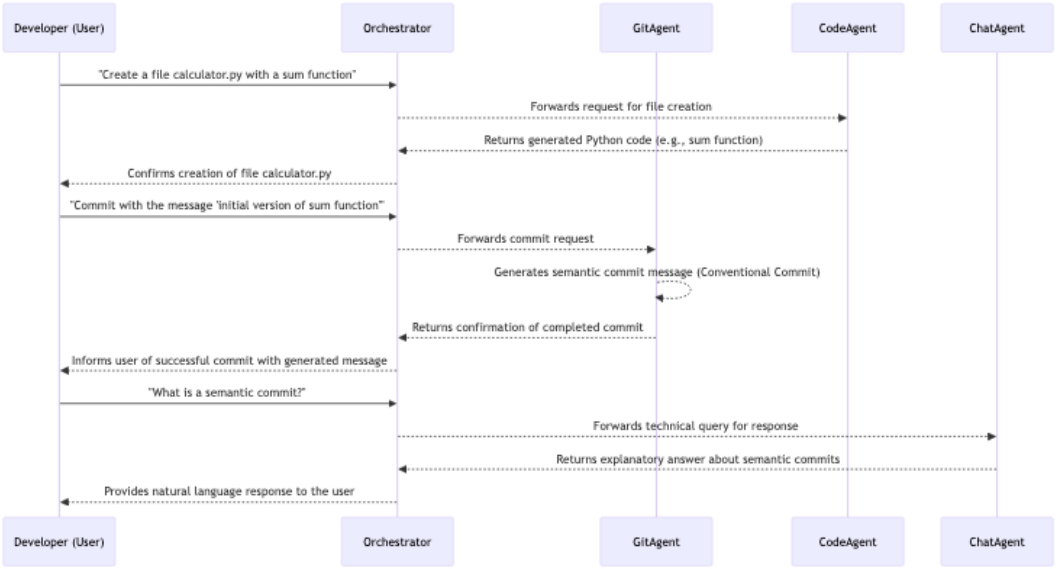

Figure 2. Interaction Workflow

3.3. Distinctive Features

OrchestraAI stands out from traditional generative AI models through several key capabilities Sapkota et al. [2025]:

- **Dynamic Reasoning and Planning**: Capable of interpreting complex user goals, decomposing tasks into actionable steps, and dynamically re-planning actions in response to user feedback and contextual changes.
- **Persistent Memory and Context Management**: Retains context across multiple interactions, enabling coherent, context-aware responses throughout the software development lifecycle.
- **Tool Use and External Interaction**: Equipped to invoke external APIs, tools, and perform operations such as code execution, version control actions, and automated testing, extending capabilities beyond text generation alone.
- **High Autonomy with Human-in-the-Loop**: While offering significant autonomy in task execution, OrchestraAI explicitly integrates iterative human feedback, balancing automation with user oversight.
- **Modular Extensibility**: Its multi-agent architecture allows for easy integration of additional specialized agents, enhancing scalability, maintainability, and adaptability to future development needs.

3.4. Implementation and Deployment

OrchestraAI is implemented using modern Python tooling, particularly using the LangChain framework for agent orchestration and model integration Narajala and Narayan [2025].

- **Backend Technologies**: Python 3.8+, LangChain with Ollama (using the model qwen3:14b) and regular expression libraries for task interpretation. The directory structure is represented in Figure 3.

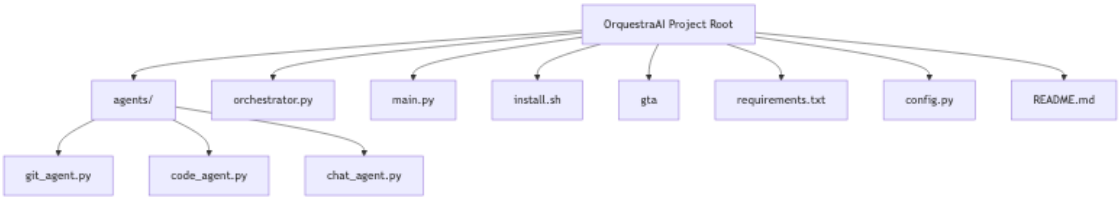

Figure 3. Directory Structure

- **Installation and Deployment**: The system provides cross-platform installation and launchers. On macOS/Linux, an automated script (install.sh) configures a local shell alias (gta) and can optionally install a system-wide launcher in /usr/local/bin/gta. On Windows, a Command Prompt launcher (gta.cmd) is included. These entry points call the same Python program (main.py), enabling seamless integration into existing development workflows. Figure 4 represents this workflow.

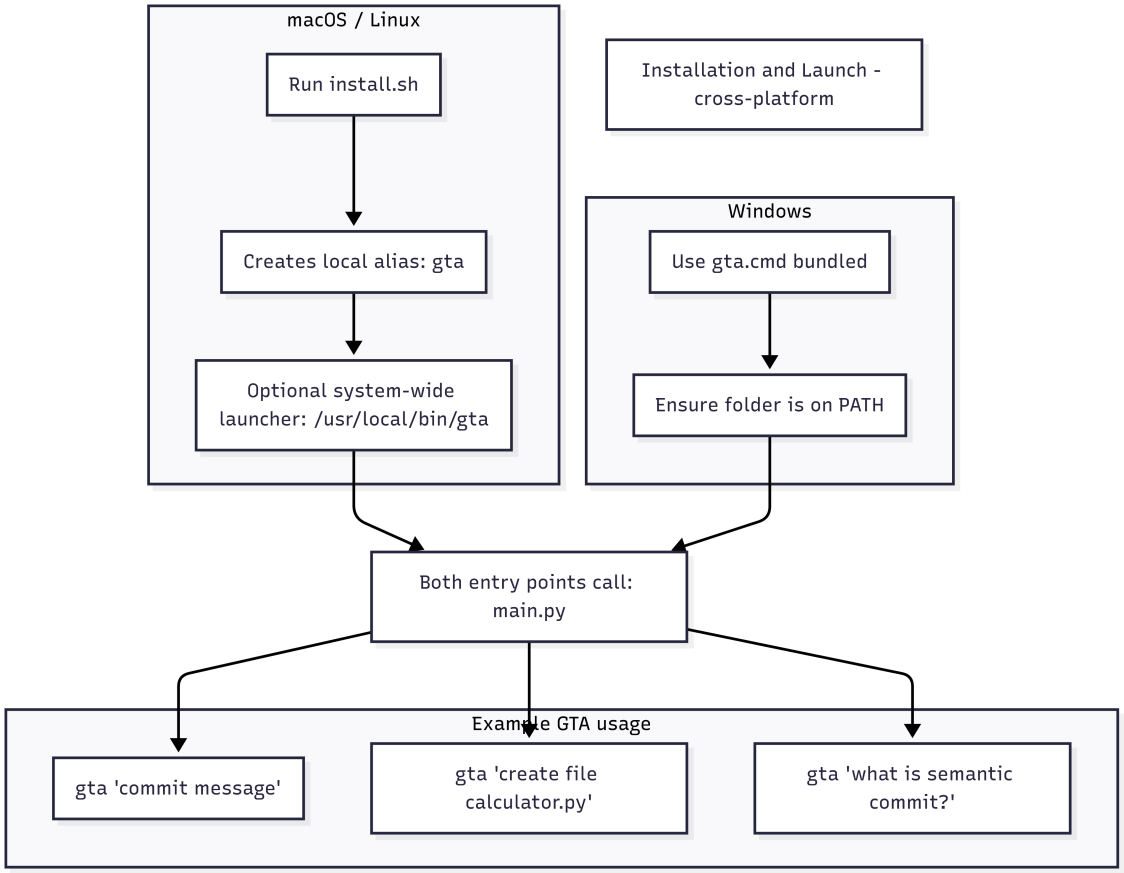

Figure 4. Implementation and Deployment

3.5. Usage Scenarios

Typical usage scenarios of OrchestraAI demonstrate its practical utility.

- **Semantic Commit Generation**: Developers describe their code changes conversationally, and GitAgent automatically generates standardized descriptive commit messages.
- **Automated Test Generation**: CodeAgent creates unit and integration tests based on existing code or specified functionality, accelerating test-driven development practices.
- **Project Structure Setup**: CodeAgent rapidly scaffolds project structures, helping developers in setting up initial boilerplate code, dependencies, and folder hierarchies.
- **Technical Documentation and Support**: ChatAgent provides on-demand documentation snippets, best-practice advice, and general programming assistance, significantly reducing research time.

In conclusion, OrchestraAI exemplifies a modern and interactive approach to software development, utilizing human-AI collaboration to maximize productivity, maintainability, and adaptability within software engineering teams.

4. Results and Discussion

This section presents the qualitative evaluation of OrchestraAI and discusses its capabilities and limitations. The evaluation focused on three scenarios: (i) code generation from natural language prompts, (ii) execution of Git version control operations based on user instructions, and (iii) multi-turn conversational interactions for iterative refinement. All tests were conducted in a controlled environment, with OrchestraAI integrated into the local file system and Git, emphasizing descriptive and interpretative analysis rather than quantitative benchmarking.

In the code generation tasks, OrchestraAI was able to generate syntactically correct and coherent code in response to clear and simple user requests, such as creating a Python function to compute a sum. These results align with the model's capabilities derived from programming-focused LLMs, demonstrating practical utility in generating boilerplate and functional code snippets. However, when faced with requests that were underspecified or involved more complex logic, the system produced incomplete or partially accurate outputs, indicating a tendency to make assumptions in the absence of explicit user guidance. Figure 5 illustrates a practical usage example of OrchestraAI, showing a developer prompt requesting the creation of a FastAPI project along with the corresponding files and project structure automatically generated by the system.

For version control operations, OrchestraAI successfully interpreted high-level user commands to perform Git-related tasks, including repository initialization, file staging, and commit creation with user-specified messages. This confirms the system's integration with development tools and its potential to automate routine versioning operations. Nevertheless, advanced Git functionalities, such as conflict resolution and branching strategies, were not evaluated, and the system currently performs reactive operations without semantic summarization of code changes.

The conversational interface of OrchestraAI proved effective in facilitating iterative development, allowing users to refine outputs and request modifications through natural

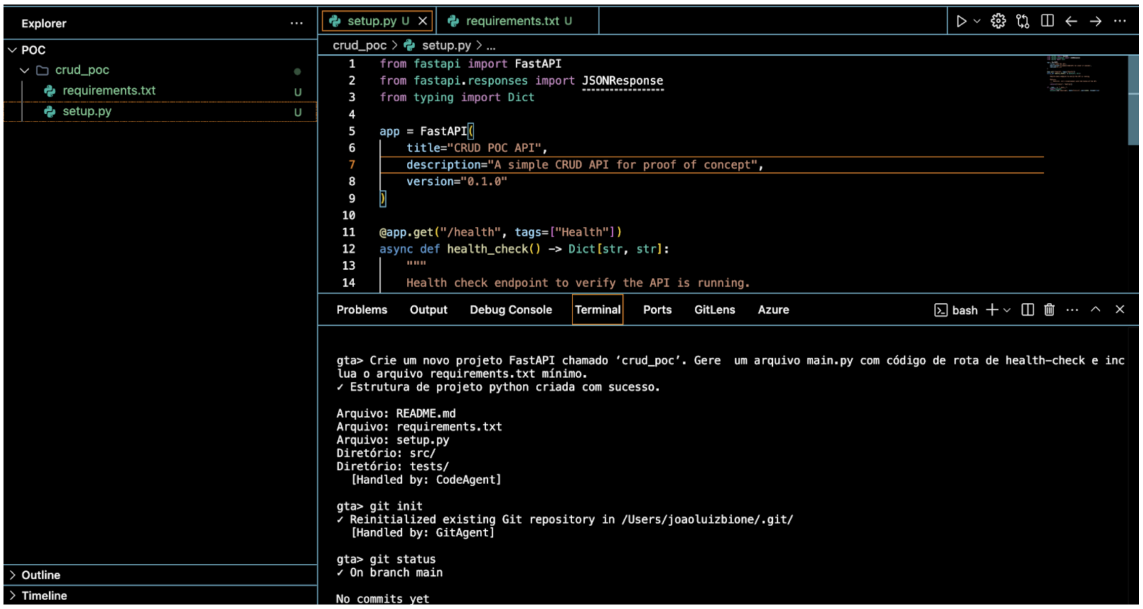

Figure 5. An OrchestraAI usage perspective.

language interactions while maintaining context over shorter sessions. This aligns with the goal of providing a co-developer experience within the user's environment. However, the system exhibited limitations in managing extended conversations, reflecting the finite context window of the underlying LLMs and highlighting the need for mechanisms that support long-term memory and context management.

Overall, the qualitative evaluation demonstrates that OrchestraAI can enhance developer productivity by automating basic programming and version control tasks through natural language interaction. Nonetheless, further work is needed to address current limitations, including the incorporation of quantitative evaluations for benchmarking, the implementation of advanced task decomposition and planning, improvements in context retention for extended sessions, and the integration of automated code execution and testing to increase reliability. Additionally, exploring enhanced user interface elements and robust security mechanisms will be essential for deploying OrchestraAI effectively in professional software development environments.

5. Threats to Validity

This study is limited by its qualitative evaluation methodology, which focused on descriptive analysis without quantitative benchmarking. As a result, it was not possible to objectively measure OrchestraAI's impact on developer productivity, task completion time, or code quality, limiting the generalizability of the observed outcomes. Additionally, the evaluation scenarios involved small, well-defined tasks within controlled environments, which may not fully reflect the complexity and variability of real-world software development workflows.

Another threat to validity concerns the scope of functionalities tested, as advanced Git operations, large codebase management, and complex multi-step task orchestration were not explored in the current implementation. Moreover, the reliance on the context window of the underlying language models may affect the system's ability to maintain coherence during extended interactions, potentially limiting usability in longer development

sessions. Future work will address these limitations through systematic quantitative studies, user-centered evaluations in diverse settings, and the integration of advanced context management and security safeguards to ensure robust deployment in professional environments.

6. Conclusion

In conclusion, this work introduced OrchestraAI, a multi-agent AI assistant tailored for software development. Our motivation was to create an AI agent that operates beyond simple chat exchanges – one that can interpret developer requests and take actions such as creating code files, editing content and performing version control commands. We addressed the need for a cost-effective and developer-centric solution by designing OrchestraAI to run locally with open-source LLMs while still allowing integration with more powerful paid LLM APIs if the user requires. This flexibility ensures that professional developers can adopt AI assistance on their own terms, maintaining privacy and control over their development workflow.

The results of our initial tests are promising. OrchestraAI successfully generated syntactically correct code from natural language prompts, carried out Git operations like commits from high-level instructions, and engaged in multi-turn dialogues to refine outputs based on user feedback. These scenarios demonstrate the potential of having an AI partner embedded in the development environment — it can save time on boilerplate coding, automate routine tasks, and help manage project state through conversational commands. Developers interacted with the system in a conversational loop, receiving immediate, context-aware support that goes far beyond what static code generation tools provide.

However, this project also revealed several challenges and limitations. Agent performance can be affected by more complex tasks or very long interaction sessions, partly due to the inherent limits of the context length of LLMs. It currently excels at well-defined, small-scale tasks, but handling large codebases or intricate multi-step operations will require further enhancements. There is also a need for a rigorous quantitative evaluation of OrchestraAI's impact on developer productivity and code quality, as our assessment so far has been qualitative. Ensuring robustness and security is vital, too. Since OrchestraAI can execute system commands and write to files, careful safeguards and user oversight are necessary to prevent unintended actions.

Despite these challenges, OrchestraAI illustrates a compelling use case for AI in software engineering. It shows that with a careful orchestration of specialized agents and human-in-the-loop design, an AI system can effectively collaborate with a developer. We believe that this approach can be extended and improved: future work will focus on integrating long-term memory for better context retention, more advanced planning algorithms for complex task decomposition, and automated code testing/verification to increase trust in the agent's outputs. In addition, expanding the range of supported developer tools and programming languages will make the system more versatile.

Ultimately, OrchestraAI represents a step toward more powerful and autonomous developer assistants that remain under the developer's guidance. By being cost-flexible and adaptable, it lowers the barrier for individuals and teams to adopt AI in their daily programming activities. With continued refinement, such AI agents could become an invaluable part of the professional developer toolkit – increasing productivity, providing on-demand expertise, and enabling a new mode of human-AI collaboration in software

development.

References

Y. Cao, S. Li, Y. Liu, Z. Yan, Y. Dai, P. S. Yu, and L. Sun. A comprehensive survey of ai-generated content (aigc): A history of generative ai from gan to chatgpt. *J. ACM*, 37 (4):111:1–111:44, 2024.

M. e. a. Chen. Evaluating large language models trained on code. *arXiv*, 2021. doi: 10.48550/arXiv.2107.03374. URL `https://arxiv.org/abs/2107.03374`. Trabalho seminal (Codex) demonstrando geração de código funcional a partir de descrições em linguagem natural e resultados em benchmarks como HumanEval.

V. S. Narajala and O. Narayan. Securing agentic ai: A comprehensive threat model and mitigation framework for generative ai agents. *arXiv preprint arXiv:2504.19956*, 2025.

L. Ning, Z. Liang, Z. Jiang, H. Qu, Y. Ding, W. Fan, X. yong Wei, S. Lin, H. Liu, P. S. Yu, and Q. Li. A survey of webagents: Towards next-generation ai agents for web automation with large foundation models. *arXiv preprint arXiv:2505.23350*, 2025.

L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Beam, C. Keller, A. Madan, W. Agarwal, J. Schulman, M. Chen, T. Brown, et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

R. Sapkota, K. I. Roumeliotis, and M. Karkee. Ai agents vs. agentic ai: A conceptual taxonomy, applications and challenges. *arXiv preprint arXiv:2505.10468*, 2025.

J. Xu, Z. Li, W. Chen, Q. Wang, X. Gao, Q. Cai, and Z. Ling. On-device language models: A comprehensive review. *arXiv*, 2024. doi: 10.48550/arXiv.2409.00088. URL `https://arxiv.org/abs/2409.00088`. Revisão que destaca benefícios de execução local: privacidade, menor latência, operação offline e redução de custos de uso.