AIPyCraft: AI-assisted software development lifecycle for 6G blockchain oracle validation

Antonio M. Alberti[1,*,†], Alexis V. de A. Leal[2,†], Ariel Galante Dalla-Costa[2,†] and Cristiano Bonato Both[2,†]

[1]*School of Computer Science, University of Leeds, United Kingdom*

[2]*Graduate Program in Applied Computing, University of Vale do Rio dos Sinos (UNISINOS), Brazil*

Abstract

The growing interest in applying Artificial Intelligence (AI) to software engineering has accelerated since the release of ChatGPT 3.5 in 2022. This paper investigates how Large Language Models (LLMs) support applications' modular development and testing. We introduce *AIPyCraft*, a novel AI-assisted framework that facilitates the end-to-end lifecycle of software projects. Our approach leverages Google Gemini 2.5 Pro model to generate, correct, and manage software components within a semi-automated and incremental workflow. *AIPyCraft* enables project creation, environment setup, error correction, and feature evolution in an integrated manner. We develop and test a blockchain-based Oracle component designed for 6G wireless network environments, i.e., a complex, real-world scenario that demands secure data integration and modular extensibility. Preliminary experiments demonstrate *AIPyCraft's* potential to accelerate small-scale software project development through an "understand-by-building" methodology. Our findings show that using an LLM to generate effective TOML jobs for Off-chain 6G functions is feasible, with an average of 1.05 iterations to correct the TOML code and mean experiment time of 27.8 seconds.

Keywords

AI-assisted coding, software engineering, automated code generation, modular software development

1. Introduction

The release of ChatGPT 3.5 on November 30, 2022, marked a significant milestone in the evolution of Large Language Models (LLMs) for code generation tasks [1]. The performance and accessibility introduced in version 3.5 sparked widespread interest across industry, academia, and the broader developer community [2]. Initial experiments commonly involved short interactions to assess the model's ability to generate and manipulate source code. However, more in-depth exploration revealed practical limitations: (i) iteratively prompting the model, (ii) copying generated code, and (iii) manually testing outputs can be time-consuming and cognitively demanding. Moreover, as projects grow in complexity, i.e., requiring multiple files,

Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC. Koblenz, Germany, June 10-13, 2025

*Corresponding author.

†These authors contributed equally.

a.m.alberti@leeds.ac.uk (A. M. Alberti); alexisleal@edu.unisinos.br (A. V. d. A. Leal); costaariel@edu.unisinos.br (A. G. Dalla-Costa); cbboth@unisinos.br (C. B. Both)

interdependent modules, and isolated virtual environments, developers frequently encounter challenges related to environment configuration, dependency management, and scalability. These barriers have highlighted the need for more structured and automated approaches to support the development of modular, testable, and maintainable software systems using LLMs.

Given these challenges, a more fitness approach would involve using autonomous agents capable of directly generating, testing, and refining code, eliminating the need for manual copy-and-paste operations and enabling project-level validation beyond isolated files. For such solutions to be practically applicable, it is essential that new features can be incrementally added and seamlessly integrated into previously validated components [3]. An incremental development strategy plays a key role in maintaining manageable levels of complexity while allowing iterative refinement. In real-world scenarios, developers begin with a general idea of the desired functionality and benefit from evolving the system through progressive modifications rather than attempting to define and fully implement the entire solution up front. Intelligent software systems that support human-in-the-loop development, combining the generative capabilities of Artificial Intelligence (AI) with human guidance, offer considerable value [4].

Recent studies in AI-assisted software development have explored the integration of LLMs into stages of the engineering workflow, including code generation [5, 6], error correction [7], configuration validation [8], and test automation [9]. While these efforts have yielded promising results, most solutions are limited in scope, e.g., focusing on narrow tasks rather than supporting the complete software lifecycle. Typical limitations include the absence of modular project design, lack of orchestration across components, and insufficient handling of environment setup or integration with virtual environments. Moreover, tools that embed LLMs into developer environments lack structured control over incremental project evolution and rely heavily on manual oversight [4]. Although techniques can be applied to emerging domains such as blockchain-enabled Sixth Generation Networks (6G) components, such as decentralized oracles and smart contract-based network automation, no comprehensive solution in the literature addresses these challenges in this context. These gaps highlight the need for holistic approaches that enable coherent, end-to-end development workflows, supporting iterative refinement, automation, and scalability.

This paper explores the subject of the AI-assisted coding lifecycle from a practical perspective, trying to answer the question: *Could LLMs help with the modular development and testing of 6G components?* Our central hypothesis is that LLMs can assist in the incremental and modular development of projects with virtual environments. Moreover, LLMs can help automate tasks such as code generation, virtual environment preparation, debugging, incremental feature addition, and project structuring. AI-powered tools can offer new paradigms to help developers with the growing complexity of software development while testing new ideas "*understanding by building*" prototypes experimenting with new features and directions on new developments. We present the *AIPyCraft*[1], an open-source collaborative project development assisted by LLMs, integrating rapid prototyping with a virtual environment for automated code running and checking. *AIPyCraft* is a first-look proposal for human-machine collaboration in software development, exploring how AI Application Programming Interfaces (APIs) are integrated to generate, manage, correct, and improve code iteratively. Moreover, *AIPyCraft* creates an entirely

[1]Source code available at: https://github.com/antonioalberti/AIPyCraft

new project, prepares its virtual environment, runs the project from its main program, collects the obtained results, corrects the components according to errors, and adds new features to a created project. The main contributions of this paper are:

- **Open-source development automation:** the proposal uses LLM APIs to generate code, manage environments, iteratively correct errors, and integrate versions, offering a single-person end-to-end solution.
- **A software component correction tool:** corrects code based on existing running errors. The prompts focus on preserving existing functionality while correcting a failing module.
- **AI-assisted development and testing:** the analysis is performed on a blockchain-based Oracle component designed for 6G wireless network environments.

Our findings indicate that using an LLM to generate effective TOML jobs in the domain of Off-chain 6G functions, is feasible with a satisfactory number of interactions (1.05 average interactions) and a reasonable time duration (27.80 seconds). The remainder of this paper is structured as follows. Section II presents the fundamental background, including our 6G development application. Section III presents the related work. Section IV describes our proposed agent architecture. Section V presents a proof of concept, outlining how *AIPyCraft* was evaluated to support the central hypothesis of the paper. Section VII concludes the study.

2. Background

Deep Learning has emerged as a dominant reference in various subfields of AI, leveraging artificial neural networks to model complex data distributions and achieving significant breakthroughs in applications such as speech recognition, computer vision, and Natural Language Processing (NLP). A pivotal advancement in NLP was Vaswani et al.'s [10] introduction of the Transformer architecture in the seminal work *"Attention Is All You Need"*. This architecture innovation laid the groundwork for the development of LLMs [11], such as Google Gemini[2], OpenAI ChatGPT-4o[3], DeepSeek[4], and Anthropic Claude[5], which are trained on extensive texts and demonstrate remarkable proficiency in generating, interpreting, and manipulating natural language in a human-like fashion. Adopting LLMs has influenced various domains, including software engineering, education, and content generation. This adoption underscores their potential to augment productivity and enable novel forms of human-AI collaboration.

The growing complexity of software development has driven the adoption of technologies to enhance the efficiency and quality of the development lifecycle [12]. LLMs have attracted attention for their remarkable ability to understand and generate source code. While initially applied to machine translation, text summarization, and image generation tasks, LLMs have been integrated into software engineering workflows [13]. Their capabilities extend to synthesis, autocompletion of code snippets, and support for testing, debugging, and documentation, positioning them as tools in development environments. The integration of LLMs into software development workflows is facilitated through APIs, offering a programmatic interface for

[2]Google Gemini. Available at: https://gemini.google.com
[3]OpenAI ChatGPT-4o. Available at: https://openai.com/chatgpt
[4]DeepSeek. Available at: https://www.deepseek.com
[5]Anthropic Claude. Available at: https://www.anthropic.com/index/claude

accessing the models' capabilities [5]. These APIs allow developers to send prompts or input queries to the LLM and receive generated text responses, which can be parsed, transformed into source code, and incorporated into local projects within an Integrated Development Environment (IDE). Through this mechanism, LLMs assist in various stages of the software engineering process, including automated code generation, unit test creation, and debugging support, thereby contributing to increased development productivity and reduced manual effort [14].

Prompt engineering has become a fundamental interface for human-AI interaction, particularly in LLM-assisted software development. To fully harness the potential of LLMs, it is crucial to devise effective strategies for designing and optimizing prompts. High-quality prompt design is vital in improving generated content's accuracy, relevance, and utility, enhancing the efficiency and reliability of AI-assisted development workflows. By carefully structuring input queries and layering them with contextual instructions, developers can guide the model's behavior to achieve more predictable and purposeful outputs. Moreover, managing virtual environments represents another key aspect of LLM-assisted software development. A virtual environment encapsulates a dedicated interpreter, project-specific libraries, and binaries, ensuring isolation from environments and system-wide installations. This isolation is essential for maintaining reproducibility, avoiding dependency conflicts, and supporting modular development practices. Integrating LLMs with virtual environments enables robust and streamlined development workflows, as the models can assist in code generation, environment setups, and configuration. In more advanced scenarios, LLMs may be employed to identify and solve package conflicts, enhancing the automation and reliability of the software development process.

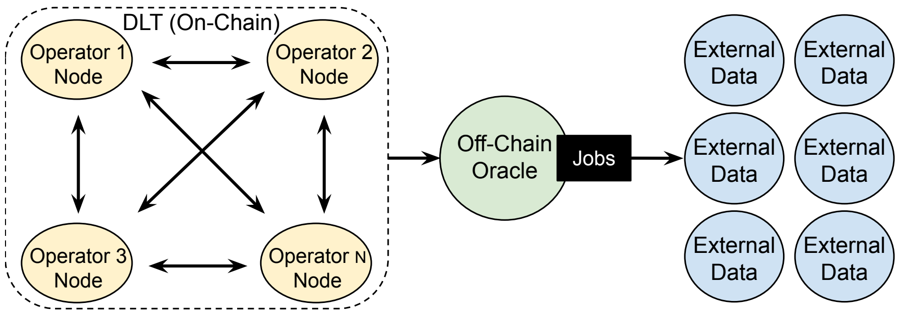


Figure 1: On-chain, Off-chain, and external data.

Adopting LLMs across domains such as software engineering becomes relevant to emerging domains such as 6G network architectures, which demand programmable, resilient, and autonomous systems. One application scenario involves blockchain-based oracle components [15], which secure bridges between Off-chain data sources and smart contracts deployed in 6G-enabled environments [16]. These oracles enable mission-critical applications, such as cognitive spectrum management, smart cities, and industrial, to access verified real-time information while addressing key challenges such as scalability, interoperability, and latency. Combining LLM-assisted development with 6G-oriented architectures represents a promising research frontier, enabling the rapid prototyping of modular and trustworthy components for next-generation wireless systems. Oracle components in the context of 6G networks act as secure middleware between Off-chain data sources and smart contracts operating within blockchain-based systems. These oracles are essential for applications that rely on real-world data, such as dynamic spectrum management, decentralized authentication, and trustworthy telemetry. They provide an interface for data acquisition, verification, and delivery of smart contracts, ensuring the integrity and reliability of network services. Figure 1 illustrates a conceptual oracle architecture integrating a 6G network with a blockchain layer. The architecture is composed of three core components: (i) a Smart Contract Interface operating on a Distributed Ledger Technology (DLT), defining and enforcing On-chain validation and storage rules [17], (ii) an Off-chain Oracle, which manages job execution, data processing, and the coordination of trust policies [15], and (iii) an External Data Layer [18], consisting of distributed data sources that provide real-time information to be retrieved, verified, and delivered by the oracle.

3. Related Work

Recent advances in AI-assisted software development have increasingly focused on integrating LLMs into stages of the software engineering lifecycle. Works have emerged targeting tasks such as code generation, testing, error correction, and configuration management. This section discusses the contributions of these works and highlights how each relates to *AIPyCraft*.

Liu et al. [6] proposed PromptV, a collaborative multi-agent framework for Verilog code generation, where LLMs specialize in hardware description. Although the work emphasizes coordination among agents, it lacks mechanisms for managing execution environments or addressing the broader development lifecycle regarding testing and integration. PromptV focuses only on isolated code-generation tasks. In contrast, *AIPyCraft* supports iterative construction and orchestration of the lifecycle beyond the code synthesis. A complementary direction is explored in LLMSecConfig [8], applying retrieval-augmented generation techniques to detect and remediate misconfigurations in containerized environments. This solution incorporates an automated correction pipeline and highlights the potential of LLMs in infrastructure-level validation tasks. Nonetheless, its scope focuses on configuration-level errors and does not encompass modular software design or lifecycle integration. In this context, *AIPyCraft* expands this mode by integrating code generation with modular assembly and tracking artifacts.

Plein et al. [7] investigate test automation based on natural language input, employing ChatGPT and CodeGPT to generate test cases from bug reports. Their results reinforce the feasibility of leveraging LLMs for targeted development tasks. This investigation shows the viability of generating tests from natural language, but it lacks integration with modular codebases or version-controlled pipelines, which *AIPyCraft* enables through structured orchestration. In a similar context, Nettur et al. introduced Cypress Copilot [9], offering an AI assistant for generating end-to-end test scripts in Web applications using Behavior Driven Development (BDD) techniques. The tool generates structured and runnable code snippets by adopting few-shot prompting with GPT-4o. While it effectively guides code creation, it does not address component orchestration or project-level modularity. In contrast, *AIPyCraft* extends beyond isolated test generation, offering modular construction and integration across the development workflow.

Cline [4] is a project initiative integrating Claude 3.5/3.7 into the VSCode IDE, facilitating autonomous, agent-based development workflows. Cline showcases how LLMs can be embedded into the developer's IDE to execute commands, edit files, and manage context with human oversight through manual approval mechanisms. Unlike *AIPyCraft*, which emphasizes modularity and full-lifecycle project construction, Cline operates within existing codebases, offering flexibility but lacking structural guidance. Moreover, Cline does not target 6G network code generation projects or deployment contexts. Table 1 reinforces the position of *AIPyCraft* as a framework that unifies prompt engineering, modular orchestration, and lifecycle support. The comparison includes key dimensions such as support for the development lifecycle, execution within virtual environments, automated error correction, component-level structuring, modular architecture, and relevance to 6G networks. While existing approaches tend to cover some of these aspects in isolation, only *AIPyCraft* integrates them as a system.

Table 1
Comparison of *AIPyCraft* with related work.

Articles/Projects	Complete Lifecycle	Virtual Environment	Error Correction	Components	Modularity	6G
Articles						
Liu et al. [6]	✗	✗	✗	✓	✓	✗
Ye et al. [8]	✓	✓	✓	✓	✓	✗
Plein et al. [7]	✗	✗	✓	✗	✗	✗
Nettur et al. [9]	✗	✗	✗	✓	✓	✗
Projects						
Cline [4]	✓	✓	✓	✓	✓	✗
AIPyCraft (this work)	✓	✓	✓	✓	✓	✓

4. *AIPyCraft* Architecture

Leveraging LLM capabilities, *AIPyCraft* supports the creation and management of software solutions in an automated and modular fashion, using an LLM API to help users generate, manage, and run code solutions. The interaction with the tool occurs through a simple menu of options. Moreover, *AIPyCraft* manages multiple stages of the open-source project lifecycle, including creating, loading, running, and updating code. Figure 2 depicts our AI agent. *AIPyCraft* contains three main abstractions: (i) Solution, (ii) Components, and (iii) Dispatcher. A Solution is an abstraction representing a simple open-source project with multiple Components. Components represent individual code files within a Solution Folder. An abstraction of a component contains methods to execute the component and convert it to and from a dictionary format. The Dispatcher holds the main menu of *AIPyCraft*, managing the lifecycle of solutions, including creating, saving, loading, preparing virtual environments, and running them. Through the Dispatcher, human users interact with the tool, selecting the actions and giving inputs that complement pre-defined tuned prompts.

One of the main characteristics of *AIPyCraft* is the solution management of LLMs using APIs. In this context, *AIPyCraft* has: (i) `AIConnector` to analyze each solution component remotely by employing an LLM, (ii) `SolutionCreator` to create new solutions and their components using LLM assistance, (iii) `SolutionLoader` to load existing solutions from folders, (iv) `SolutionImporter` to import external folders as solutions, and (v) `SolutionImporter`

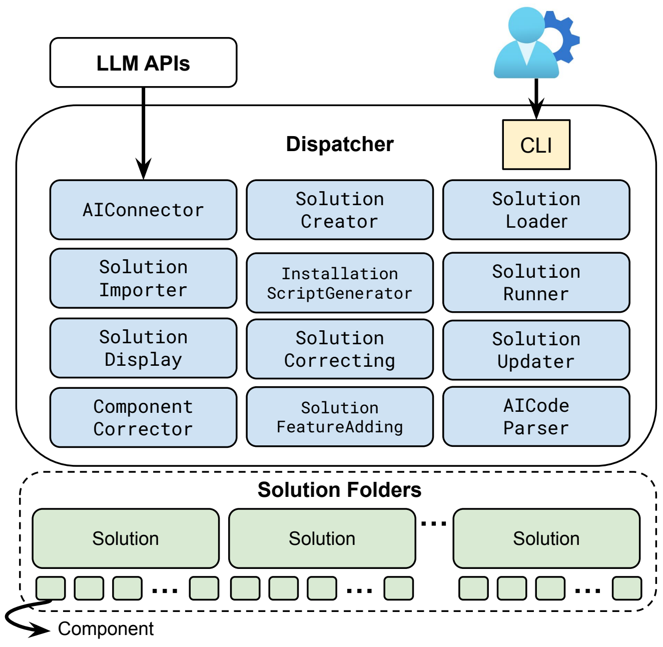


Figure 2: *AIPyCraft* architecture.

to import repositories as a solution inside the tool. After creating a solution, *AIPyCraft* offers: (vi) `InstallationScriptGenerator` function to install solution packages according to their dependencies and avoid conflict, (vii) `SolutionRunner` to run solution calls and automatically collects possible errors, (viii) `SolutionDisplay` to show solution details and code, (ix) `SolutionCorrecting` (x), `SolutionUpdater` (xi), `ComponentCorrector` (xii), and `SolutionFeatureAdding` to correct and improve a solution, and (xiii) `AICodeParser` to extract, save, and detect the language of code blocks from AI responses.

Figure 3 illustrates the workflow of *AIPyCraft* usage, showing the main steps while collaboratively developing a new open-source project. `SolutionCreator` manages to create solutions with multiple components/files and includes human approval in the feedback loop. This process creates isolated environments within a solution's directory, ensuring each solution has its contained development environment with necessary dependencies installed. The user inputs a solution name and semantic description. After, LLM generates a plan for components needed, such as classes, accessory files, text, and main program. Each component is created individually with AI assistance. Solutions are saved in the specified directory and rejected solutions are stored so users can adjust the creative process in partnership with the LLM.

The process begins by prompting the user to describe a desired new feature. *AIPyCraft* constructs a detailed prompt for the LLM API, including the solution's name, component details, and the new feature description. *AIPyCraft* checks for a valid solution and locates the required main program component. Moreover, *AIPyCraft* has an error correction system that automatically fixes code/scripts when solutions fail during execution. In this context, *AIPyCraft* works by (i) generating targeted prompts for AI code correction, (ii) analyzing each

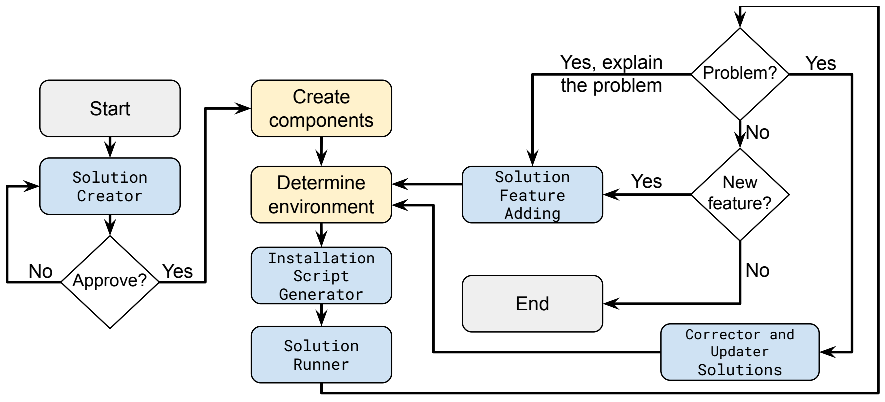


Figure 3: Typical sequence of actions done by a user in *AIPyCraft*.

solution component remotely by employing an LLM, (iii) handling code updates with proper file I/O operations when required, (iv) trying to maintain naming conventions and parameter consistency via prompt engineering, and (v) ensures proper imports to the main program. Therefore, *AIPyCraft* writes the new code to the appropriate component file if updates are found, ensuring its name remains unchanged. Finally, the process of code construction concludes when no additional execution errors are detected or the user declines further feature additions.

5. Proof-of-Concept

We test AIPyCraft with Google Gemini 2.5 Pro to correct a TOML script named `config.toml`. This TOML script creates and runs the Off-chain Oracle using the Chainlink tool that requests jobs for 6G based on the D6G architecture [16]. We automated the execution of multiple independent test runs (trials) for each Chainlink direct request job specification tested, as illustrated in Figure 4. An initialization script is performed for each trial, passing the target solution name and base path, to ensure that a TOML Script Tester starts from a known consistent state. The Chainlink direct request job specification script is cleared before each test.

Our multi-trial test script runs a Python-developed `tester.py`, providing it with the maximum number of internal correction loops allowed (-LoopsValue), a unique identifier for the trial (–run-id), the target solution name (–solution-name), the base path (–solutions-base-path), and the specific correction instructions (–correction-prompt) to be used by the AI during that trial's correction phase. A total of 20 trials have been performed for each evaluated Chainlink direct request job specification script. Table 2 shows our test configuration parameters. In each trial, `tester.py` invokes *AIPyCraft* to load a TOML script tester containing the `config.toml` Chainlink job specification and two human-developed testing programs. Two scenarios have been evaluated: Scenario 1, which only verifies the syntax of the TOML script using the `tomli` Python library, and Scenario 2, which provides successful Job script testing. In Scenario 1, success means a call to `tomli.load(f)` returned without syntax errors. In Scenario 2, the TOML script tester sends a new `config.toml` version created by the LLM to a Chainlink node running on a virtual machine. A Job is created with the TOML script received and then run. In

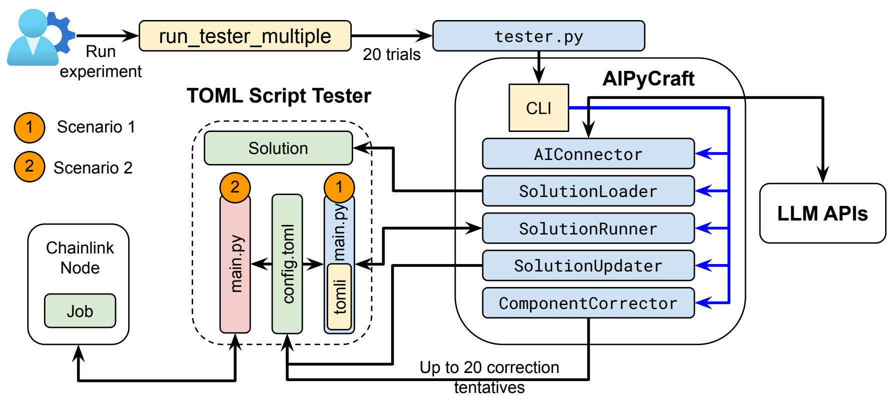


Figure 4: Experimental trials using *AIPyCraft*, Gemini 2.5 Pro, and Chainlink node.

this case, success means that the LLM generated a TOML script that enabled the creation of a Job in the VM Chainlink node and ran without errors.

Table 2
Python Tester Script Parameters for Automated Multi-Trial Testing.

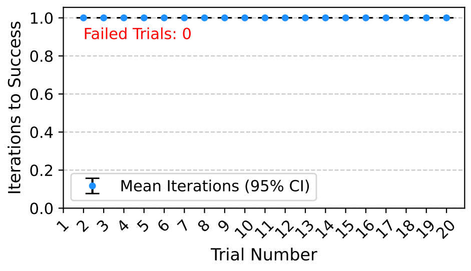
Parameter Name	Value/Setting	Description
`-LoopsValue`	20	Maximum number of internal correction loops allowed per trial
`-run-id`	Unique per trial	A unique identifier assigned to each individual test trial
`-solution-name`	Target specific	The name of the target solution being tested in the trial
`-solutions-base-path`	Path specific	The base directory path where solutions are located
`-correction-prompt`	Instruction specific	AI provides the correction instructions of a trial

Figure 5a shows the number of interactions required to correct Chainlink direct request Job syntax using `tomli` Python package, i.e., Scenario 1. In all trials, our solution successfully corrected the Job script. The mean number of interactions was only 1, meaning the first correction proposed succeeded. Moreover, Figure 5b shows the total time spent in each trial. The mean Job correction time was 27.80 s, with a minimum value of 19.09 s and a maximum of 35.20 s. The results highlight that using an LLM API to create useful TOML jobs for a Chainlink node in the context of off-chain 6G functions is possible with acceptable time.

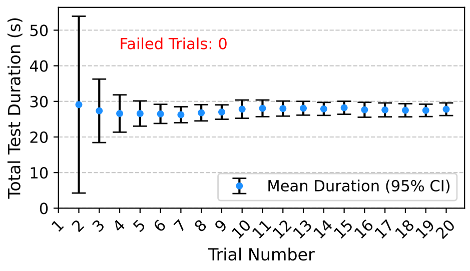
Figure 6a shows the results for Scenario 2, which evaluated the TOML script more thoroughly. The mean number of correction interactions was 1.05 since two were required in one trial. Figure 6b shows the time needed in Scenario 2. The mean correction time was 37.61 s, with a maximum time equal to 74.67 s and a minimum of 27.61 s. The results indicate that the integration of *AIPyCraft* with Gemini 2.5 Pro proved efficacious in generating TOML scripts (Jobs) capable of execution on a local Chainlink node within a reasonable timeframe.

6. Conclusion

We have integrated a new AI-assisted development tool called AIPyCraft with Gemini 2.5 Pro to successfully correct and run TOML scripts (Jobs) in a local Chainlink node. The Jobs created were able to listen to Off-chain Oracle requests within a real Chainlink implementation. Such

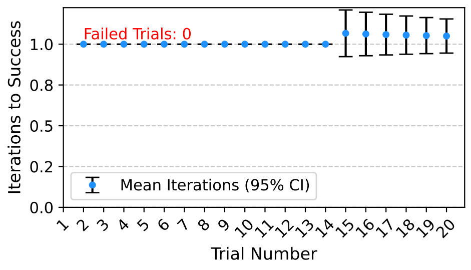


(a) Number of interactions required.

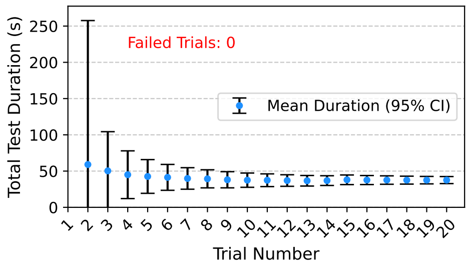


(b) Time required.

Figure 5: Syntax analysis to correct Job script with `tomli`.



(a) Number of interactions.



(b) Time required.

Figure 6: Semantic analysis to correct Job script in Chainlink node.

AI-generated Jobs are essential to a disruptive 6G (D6G) wireless mobile network proposal being developed by the authors. The Jobs will connect On-chain public smart contracts with Off-chain 6G service instances. Our experimental tests showed that the mean number of correction interactions required to run a Job created by an LLM successfully was 1.05 attempts of 20 allowed. The mean time to achieve this result was 37.61 s, which is acceptable in deploying new 6G services connected to On-chain smart contracts. In conclusion, these results prove that our solution can be successfully applied in the modular development and testing of 6G components with acceptable performance. Future work includes comparison among different LLMs, evaluation of additional AI models for multi-agent collaboration, different AI-assisted decision-making, expanding to other programming languages and environments, improving security for AI-assisted workflows, and evaluation of different purpose Chainlink Jobs for 6G.

Declaration on Generative AI

During the preparation of this work, the author(s) used Chat-GPT-4 to support the improvement of grammar and spelling checking. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

Acknowledgments

This work was conducted with partial financial support from CNPq, MCTIC/CGI.br/FAPESP through PORVIR-5G project n^o 2020/05182-3, and RNP with resources from MCTIC Grant n^o $01245.020548/2021-07$, under the Brazil 6G project.

References

[1] V. Venkatasubramanian, A. Chakraborty, Quo Vadis ChatGPT? From large language models to Large Knowledge Models, Computers & Chemical Engineering 192 (2025) 108895.

[2] A. Buscemi, A Comparative Study of Code Generation using ChatGPT 3.5 across 10 Programming Languages, 2023. arXiv:2308.04477.

[3] M. Beyazit, et al., Incremental testing in software product lines—an event based approach, IEEE Access 11 (2023) 2384–2395.

[4] Anthropic, Cline: Agentic Development with Claude 3.5/3.7, https://github.com/anthropics/cline, 2025. Accessed: March 31, 2025.

[5] X. Gu, et al., On the Effectiveness of Large Language Models in Domain-Specific Code Generation, ACM Trans. Softw. Eng. Methodol. 34 (2025).

[6] Y. Liu, et al., PromptV: Large Language Model based Multi-agent Framework for Verilog Code Generation, in: Conference on Artificial Intelligence, 2024.

[7] L. Plein, et al., Automatic Generation of Test Cases based on Bug Reports: a Feasibility Study with Large Language Models, in: International Conference on Software Engineering: Companion Proceedings, 2024.

[8] Z. Ye, et al., LLMSecConfig: An LLM-Based Approach for Fixing Software Container Misconfigurations, arXiv preprint arXiv:2502.02009 (2025).

[9] S. B. Nettur, et al., Cypress Copilot: Development of an AI Assistant for Boosting Productivity and Transforming Web Application Testing, IEEE Access 13 (2025).

[10] A. Vaswani, et al., Attention Is All You Need, in: Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017, pp. 6000–6010.

[11] H. Zhao, et al., Explainability for Large Language Models: A Survey, ACM Trans. Intell. Syst. Technol. 15 (2024).

[12] U. Durrani, et al., A decade of progress: A systematic literature review on the integration of ai in software engineering phases and activities (2013-2023), IEEE Access 12 (2024) 171185–171204.

[13] M. Debbah, Large language models for telecom, in: 2023 Eighth International Conference on Fog and Mobile Edge Computing (FMEC), 2023, pp. 3–4.

[14] J. Wang, et al., Software Testing With Large Language Models: Survey, Landscape, and Vision, IEEE Transactions on Software Engineering 50 (2024) 911–936.

[15] D. Cuellar, et al., BSM-6G: Blockchain-Based Dynamic Spectrum Management for 6G Networks: Addressing Interoperability and Scalability, IEEE Access 12 (2024) 59464–59480.

[16] A. M. Alberti, et al., Disruptive 6G architecture: Software-centric, AI-driven, and digital market-based mobile networks, Computer Networks 252 (2024) 110682.

[17] Y. Du, et al., A Novel Oracle-Aided Industrial IoT Blockchain: Architecture, Challenges,

and Potential Solutions, IEEE Network 37 (2023) 8–15.

[18] Y. Zuo, et al., A Survey of Blockchain and Artificial Intelligence for 6G Wireless Communications, IEEE Communications Surveys & Tutorials 55 (2023) 1–39.

[19] T. B. Brown, et al., Language Models are Few-Shot Learners, arXiv preprint arXiv:2005.14165 (2020).

[20] A. Pasdar, et al., Connect API with Blockchain: A Survey on Blockchain Oracle Implementation, ACM Computing Surveys 55 (2023) 1–39.