MASON - A Multi-Agent LLM Framework for No-Code Development

Muhammed Roshan Palayamkot, Kayvan Karim

Association for the Advancement of Artificial Intelligence
Heriot Watt University Dubai
mroshan5722@gmail.com, K.Karim@hw.ac.uk

Abstract

The rise of No-Code Development (NCD) has enabled citizen developers to build applications without traditional programming expertise. However, as these platforms scale to handle complex, interdependent tasks, their limitations become apparent. Large Language Models (LLMs) offer a potential solution, yet single-agent systems often struggle to manage full-stack development reliably.

This study introduces **MASON**—a **M**ulti-**A**gent **S**ystem (MAS) for **O**pen **N**o-code development framework—that coordinates specialized LLM agents via a YAML-based workflow to automate NCD tasks. The system was evaluated across four proprietary models—Claude 3.5 Sonnet, GPT-4o Mini, Gemini 1.5 Flash, and DeepSeek-Chat—using HumanEval and MBPP benchmarks to assess accuracy, execution time, and stability. MASON configurations showed improved task reliability in simpler workflows but introduced latency on more complex tasks. Additional testing with small, locally hosted LLMs revealed significant limitations, emphasizing the need for architectural redesign or model fine-tuning to support deployment in resource-constrained environments.

Code — https://github.com/mrp2003/MASON

HumanEval Dataset — https://github.com/openai/human-eval

MBPP Dataset — https://github.com/google-research/google-research/tree/master/mbpp

Introduction

No-code development (NCD) platforms have revolutionized software creation by enabling non-technical users to build applications through visual interfaces and prebuilt components (Beranic, Rek, and Hericko 2020; Sufi 2023; Martinez and Pfister 2023; ALSAADI et al. 2021). Despite their accessibility, these platforms struggle with scalability and coordination when handling complex, multi-stage workflows (Di Ruscio et al. 2022; Pinho, Aguiar, and Amaral 2023; Luo et al. 2021; Yan 2021; Bock and Frank 2021; Shen et al. 2024).

Large Language Models (LLMs) have demonstrated remarkable success in automating programming tasks (Lin, Kim et al. 2024; Touvron et al. 2023; Shen et al. 2023; Xie et al. 2023; Schäfer et al. 2023; AlOmar et al. 2024; Ma et al. 2024b,c). Their ability to understand natural language, generate code, and reason across contexts makes them promising tools for enhancing NCD. However, standalone LLMs still suffer from issues like hallucinations (Yang et al. 2023; Zhang et al. 2023; Talebirad and Nadiri 2023; Bang et al. 2023) and poor performance on long, interdependent tasks (Ma et al. 2024a), limiting their reliability in isolation.

Recent research has addressed these challenges by integrating LLMs into Multi-Agent Systems (MAS), where specialized agents collaborate, communicate, and share contextual knowledge (He, Treude, and Lo 2024; Liu et al. 2023b; Yuan et al. 2023; Talebirad and Nadiri 2023; Du et al. 2023; Liang et al. 2023a). This modular design supports task delegation—such as code generation, validation, and packaging—offering a more scalable and robust approach to NCD automation.

However, most MAS frameworks rely on high-capacity proprietary LLMs, restricting deployment in privacy-sensitive environments or regions with limited connectivity. In contexts like healthcare, education, or remote field operations, relying on external APIs may be infeasible due to legal, ethical, or infrastructural constraints. This raises a key question: *Can small, open-source LLMs support MAS-based NCD as effectively as large proprietary models?* If so, such systems could lower barriers to intelligent automation globally.

This study investigates that question by applying a MAS framework—originally designed for large models—to small, locally hosted LLMs. The results motivated two future directions: redesigning MAS frameworks for low-resource compatibility or fine-tuning smaller LLMs to function within existing orchestration strategies. To do this, we developed MASON using a YAML-based CrewAI setup with specialized agents and sequential coordination.

Background

NCD has empowered non-technical users to build applications through intuitive interfaces and prebuilt components (Beranic, Rek, and Hericko 2020; Sufi 2023; Martinez and Pfister 2023; ALSAADI et al. 2021). Despite their accessibility, these platforms often struggle to scale with complex workflows that require error handling, long-term state management, or external system integration (Di Rus-

cio et al. 2022; Pinho, Aguiar, and Amaral 2023; Luo et al. 2021; Shen et al. 2024). While cross-platform tools and blockchain-enabled solutions have extended the reach of NCDs (Curty, Härer, and Fill 2022; Solanky, Patil, and Patel 2016), usability and performance remain limited. Comparative studies show that enterprise-level readiness—particularly in areas like security, resource management, and governance—lags behind expectations (Sahay et al. 2020; Balamurugan et al. 2023). Automation through rule-based workflows and optimization models (Bahsi, Ceyhan, and Kosar 2007; Logeshwaran et al. 2023; Karamthulla, Malaiyappan, and Tillu 2023) adds complexity that non-developers often find difficult to manage, highlighting the need for intelligent, adaptive backend systems.

In response, LLMs have gained traction for automating code and content generation across a variety of domains (Wang et al. 2024; Lin, Kim et al. 2024; Touvron et al. 2023; Xie et al. 2023). Their ability to generalize language patterns and generate human-like responses makes them valuable in software automation. However, limitations such as hallucinations (Yang et al. 2023; Zhang et al. 2023), context size constraints (Peng et al. 2006), and their reliance on expensive compute infrastructure raise questions about their feasibility in scalable deployment (Aubry et al. 2024; Shabir et al. 2024). Techniques such as memory-augmented reasoning (Miret and Krishnan 2024), architectural innovations (Perez et al. 2024; Liu et al. 2023a), and optimization strategies (Sarumi and Heider 2024; Liang et al. 2023b) offer promising advances, but many of these remain theoretical or benchmark-bound. These constraints have prompted research into lighter alternatives, particularly in resource-constrained or offline environments.

One such direction is the use of MAS powered by LLMs. MAS frameworks distribute tasks across multiple specialized agents, each responsible for subtasks like prompt generation, code evaluation, or validation. This modularity improves robustness and fault tolerance, especially in workflows where complex, interdependent actions must be coordinated (Liu et al. 2023b; Yuan et al. 2023; Goonatilleke and Hettige 2022). Several MAS-based frameworks have emerged in recent literature, demonstrating varying degrees of success across benchmark and real-world environments.

Among them, MetaGPT (Hong et al. 2023) simulates a software development team using SOP-driven agents such as Product Manager, QA Engineer, and Architect. It performs decomposition of requirements into artifacts like requirement docs, design diagrams, and code, showing improvements in code executability across HumanEval and MBPP. However, it lacks support for UI or frontend code and struggles with flexibility in unstructured workflows. Similarly, CodePori (Rasheed et al. 2024) employs six collaborative agents—ranging from Dev and Finalizer to Verification agents—to generate production-ready code using prompt refinement. It achieves 89% on HumanEval but focuses heavily on structured pipelines and GPT-tier APIs, making it unsuitable for lightweight or highly interdependent workflows.

AgentLite (Liu et al. 2024) introduces a customizable, lightweight agent framework with modular reasoning blocks like PromptGen, Actions, and Memory. It has been applied in sandbox tasks like retrieval-augmented QA and image captioning, but lacks validation in end-to-end application development and real-world agent orchestration. TrainerAgent (Li et al. 2023) tackles model training pipelines using Task, Data, Model, and Server agents. It was validated on the Taobao platform in tasks like Visual Grounding and Text Classification but still requires human intervention for complex or under-specified scenarios.

Other frameworks explore structural coordination and autonomy. SoA (Ishibashi and Nishimura 2024) introduces hierarchical agents—Mother and Child—to manage task decomposition and refinement, achieving high benchmark scores but leaving communication overhead and system reliability underexplored. Talebirad and Nadiri (Talebirad and Nadiri 2023) propose a graph-structured MAS with dynamic agent addition and self-feedback, yet fail to evaluate performance under real-world constraints or ethical concerns. Similarly, LCG (Lin, Kim et al. 2024) simulates traditional workflows like Scrum and Waterfall, assigning LLM agents roles such as developer, tester, and scrum master. It reports a 31.5% improvement over GPT baselines but is evaluated only on synthetic benchmarks with limited generalizability.

Finally, He et al. (He, Treude, and Lo 2024) present an orchestration platform for agent collaboration using role-based coordination between QA Engineers and Project Managers. While conceptually robust, the framework lacks empirical validation and detailed implementation for deployment at scale, especially with smaller LLMs.

These frameworks illustrate the increasing interest in applying MAS to software automation, but most rely on large proprietary LLMs, pre-defined workflows, or fixed APIs. Their generalization to dynamic, full-stack no-code workflows—especially in low-resource environments—remains untested. This paper addresses that gap by proposing MASON: a CrewAI-based, YAML-orchestrated MAS system that coordinates small, open-source LLM agents for code generation and validation. Unlike previous systems, MASON evaluates both local and cloud-based agents under shared experimental conditions, enabling insights into reliability, scalability, and feasibility across diverse settings.

Methodology

The design and development of MASON focused on automating and streamlining complex tasks using LLM agents. The primary objective was to construct an autonomous agentic system capable of managing the full application lifecycle—from interpreting user input to generating and validating executable code—while maintaining compatibility with multiple LLM backbones.

Agentic Frameworks

To support multi-agent collaboration, several agentic frameworks were evaluated for their suitability in orchestrating LLM-based agents within a NCD context. The focus was on identifying a framework that offered ease of implementation, robust coordination, and compatibility with diverse LLMs.

Among the evaluated frameworks, AutoGen (AG) offered advanced orchestration capabilities and fine-grained

agent control but required extensive configuration and had a steep learning curve, making it less suitable for rapid deployment (Wu et al. 2023). LangChain (LC), while strong in integrating LLMs with external tools, focused primarily on tool-assisted workflows rather than native multi-agent coordination, limiting its suitability for fully autonomous agent systems (Topsakal and Akinci 2023). In contrast, CrewAI(Duan and Wang 2024) provided a lightweight, declarative architecture with YAML-based configuration, enabling native multi-agent workflows, simplified role assignment, and seamless LLM integration—making it ideal for fast prototyping and experimentation (Duan and Wang 2024; Barbarroxa, Gomes, and Vale 2024).

Ultimately, **CrewAI**(Duan and Wang 2024) was selected for its balanced feature set, clear task sequencing, and stable execution, aligning with the project's need to support multiple LLM backbones and repeated testing without orchestration overhead.

Criteria	AG	LC	CrewAI
Learning Curve	No	No	Yes
Multi-Agent Support	Yes	No	Yes
Integration Ease	Yes	Yes	Yes
LLM Compatibility	Yes	Yes	Yes
Task Sequencing Support	Yes	No	Yes
Lightweight Setup	No	Yes	Yes
Community Support	Yes	Yes	Yes
Stability in Repeated Tasks	Yes	Yes	Yes
Suitability for No-Code Tasks	No	No	Yes

Table 1: Framework Comparison for MAS Deployment

Role Assignment and Agent Initialization

MASON was structured around a collaborative, sequential workflow where LLM agents were assigned specialized roles with distinct goals, responsibilities, and execution parameters, all defined via `YAML` configuration files. Each agent played a specific part in the development pipeline, as outlined in Table 2.

Agent Name	Role Description
Input Parser	Extracts actionable details from user input.
Requirements Analyzer	Converts parsed data into a structured software specification.
Code Generator	Produces Python code matching the software spec and adhering to best practices.
Code Validator	Validates code for errors and checks compliance with unit tests.
File Output Specialist	Packages validated code into an executable Python file (`output.py`).

Table 2: Agent Roles and Their Specialized Responsibilities

Each agent's `goal` and `backstory` were fine-tuned to guide LLM reasoning. `Tasks` were mapped to agents and executed sequentially, with each output feeding into the next.

Agent roles and tasks were initialized using CrewAI's(Duan and Wang 2024) agent-task configuration system. `YAML` files defined role behavior, tools, and execution parameters. This setup enabled detailed control but introduced several challenges during initialization.

A major challenge was clearly defining task boundaries between reasoning and execution, which required extensive prompt tuning and experimentation. Early designs explored merged or split agent roles—for example, combining parsing and analysis—but these led to ambiguous responsibilities and degraded coordination. The current five-agent structure was chosen for its clarity, modularity, and consistent performance in pilot tests, striking a balance between specialization and workflow simplicity.

Collaborative Workflow Management

With agents and tasks initialized, attention shifted to designing an effective collaboration workflow where agents could operate autonomously yet cohesively. CrewAI's(Duan and Wang 2024) sequential execution model enabled controlled task handover, predictable behavior, and tight coupling between steps—ensuring smooth progression from input parsing to code generation and final output.

User Input: The process began with user-defined data—typically a software request—being input, serving as the entry point for the pipeline.

Task Sequence: Agents executed their roles in a fixed order:

Figure 1: This sequence ensured clarity in task boundaries and smooth data progression.

Communication: Task outputs were seamlessly passed between agents using task context, with CrewAI(Duan and Wang 2024) managing execution flow, retries, and error recovery in the background.

Self-Thought and Feedback: Agents were empowered with limited self-correction capabilities. If unexpected outcomes were detected, agents could re-evaluate their outputs within a *feedback loop*, enhancing workflow reliability without requiring human intervention.

To support specific agent actions—such as saving the final validated code—a set of CrewAI tools, including FileWriterTool, was integrated seamlessly into the system. The framework's LLM-agnostic design allowed agents to interact with different models via API, enabling flexible reasoning strategies and supporting comparative evaluation across LLM providers.

Tools and Technologies Used

The development process initially prioritized the use of small, locally hosted LLMs to evaluate whether a fully self-contained, offline-compatible MAS could be achieved.

Models were deployed using Ollama, an open-source platform that supports on-device inference. It also enabled compliance with data governance policies by keeping all computation and user data entirely on-device. The local models tested—including multiple versions of Qwen2.5, Gemma3, LLaMA3, and DeepSeek R1—are listed in Table 3.

Local Models (via Ollama)	Model Provider	Versions Tested
Qwen2.5	Alibaba	0.5B, 1.5B, 3B, 7B
Qwen2.5-Coder	Alibaba	0.5B, 1.5B, 3B, 7B
Gemma3	DeepMind	1B, 4B
LLaMA3.2	Meta AI	1B, 3B
LLaMA3.1	Meta AI	8B
DeepSeek R1	DeepSeek	1.5B, 7B, 8B

Table 3: Local Models

Despite its appeal, the local setup faced practical limitations. Models frequently encountered memory exhaustion, latency spikes, and produced incomplete or incoherent outputs—especially in multi-step workflows requiring sustained reasoning. While some failures stemmed from CPU-only execution, the primary issue was the limited capacity of small models to support role-specific collaboration in MASON. Quantitatively, models with 0–1.5B parameters achieved a 0% pass rate, 3–4B parameter models reached just 6%, and even 7–8B models only managed 11%, underscoring consistent failure across scales.

These results highlight that MASON's design depends heavily on the reasoning strength and output reliability of the underlying LLM. Small models failed to complete even basic tasks, pointing to the need for architectural adjustments or model-level fine-tuning for lightweight deployment.

To validate the system under stable conditions, development shifted to high-performance, cloud-based LLMs via official APIs. This enabled consistent execution, reduced failures, and allowed for direct benchmarking across diverse LLM architectures. The cloud models used in this phase are listed in Table 4.

API Models Used	Model Provider
GPT-4o Mini	OpenAI
Claude 3.5 Sonnet	Anthropic
Gemini 1.5 Flash	Google DeepMind
DeepSeek-Chat	DeepSeek

Table 4: Cloud Based Models

These models were selected to assess how architecture and provider-specific capabilities influence agent performance in a multi-agent setting. Integration was achieved using official APIs, while CrewAI(Duan and Wang 2024) managed orchestration and workflow across all setups, ensuring compatibility and consistency.

To support these evaluations, a custom framework was developed to standardize task execution and logging across different models. This was implemented using a Python script (`main.py`) that orchestrated agent interaction and metric evaluation.

The `load_tasks()` function loaded benchmark tasks from a predefined dataset using a fixed seed for consistent sampling. `crew.kickoff(inputs)` managed agent coordination and sequential task execution based on the input specification, test cases, and function name. After code generation, `run_tests()` validated the output by dynamically importing the generated script and executing the associated unit tests. Finally, `evaluate_pass_at_1()` recorded results and computed the pass@1 score across all completed tasks, providing a standardized metric for LLM performance.

The development process began with framework setup and evolved through iterative testing using preset prompts of varying difficulty. This approach progressively refined the system's ability to handle a range of task-based experiments designed to evaluate agent interaction, execution reliability, and overall MASON performance. Four distinct tasks were devised, each targeting a specific dimension of functionality, complexity, or scalability, as summarized in Table 5.

Experiment	Task Description	Objective
Simple App	Calculator with Basic UI and Logic	Validate baseline MASON functionality
Game App	Tic-Tac-Toe with Console UI and Game Logic	Assess logic handling and interaction
Real-World App	Trading App with API Integration)	Test integration capabilities
Stress Test	E-commerce Cart System (Scalability)	Evaluate system limits and recovery

Table 5: Overview of experimental tasks designed to test MASON robustness and flexibility.

Several debugging techniques were employed to support these evaluations. Trial-and-error testing was crucial early on to identify issues in prompt design and task flow. Frequent reference to CrewAI documentation helped resolve configuration questions, while detailed error logs informed improvements in prompt structure and agent responsibilities.

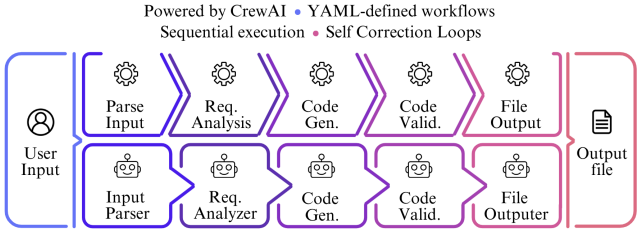

Figure 2: System Architecture

System Architecture Diagram

To encapsulate the MASON design, Figure 2 presents an overview of the complete system architecture, including agent interactions, task sequencing, and coordination logic. The workflow is structured into distinct layers—input handling, processing, and output generation—managed through CrewAI's(Duan and Wang 2024) orchestration. Each layer contributes a specific function in the pipeline, from parsing user inputs to producing validated Python code. Feedback loops ensure resilience and recovery during execution.

Evaluation

The performance and reliability of MASON was evaluated against a conventional single-agent LLM setup to determine the effectiveness of coordinated, task-specialized execution. Both architectures were tested under identical conditions, using the same inputs and evaluation criteria, ensuring that observed differences could be attributed solely to system design.

Evaluation Design and Rationale

Central to this comparison were two performance indicators. The first was **Execution Time**, which served as a straightforward measure of system efficiency — specifically, how long each system or model took to process and respond to a task. This metric highlighted underlying differences in model architecture, processing coordination, and system overhead. For MASON, execution time also captures inter-agent handovers and feedback loop costs.

The second was **Pass Rate**, evaluated using the *Pass@1 score* — a standard metric for assessing the correctness of a solution generated in a single attempt. Since both MASON and single LLM systems generated only one solution per task, this metric provided an accurate gauge of first-attempt success. The solution was validated via unit tests, and success was logged sequentially for all tasks in the test. Alongside this, fail counts were recorded to identify patterns of instability or failure under specific conditions.

The *Pass@1 score* was calculated using the following formula:

$$Pass@1 = \frac{NumberofSuccessfullyPassedTasks}{TotalNumberofTasksAttempted}$$

Together, these metrics provided a balanced view of **speed vs reliability**, allowing an informed comparison between MASON and single-agent LLMs across a range of challenges.

Evaluation Setup and Conditions

Assessing system capability requires more than just measuring performance metrics — it demands a level playing field. To isolate the effect of system design, all experiments were conducted under strictly controlled conditions. Four leading LLMs—GPT-4o Mini, Claude 3.5 Sonnet, Gemini 1.5 Flash, and DeepSeek-Chat—were tested both independently and within MASON, using identical prompts, APIs, and runtime settings. Each of MASON's agent used the same underlying LLM, ensuring any differences stemmed from architecture rather than model variance. CrewAI(Duan and Wang 2024) handled agent orchestration, task flow, retries, and inter-agent communication in all MASON runs.

In addition, smaller local models (e.g., LLaMA3, Gemma, Qwen) were tested to assess the feasibility of resource-constrained deployment. These consistently failed to generate valid or complete outputs under MASON, highlighting the need for architectural redesign or fine-tuning to support smaller models.

Task Dataset and Sampling

The choice of evaluation tasks plays a critical role in revealing the strengths and limitations of a system. For this study, the tasks were curated from two established benchmarks: **HumanEval** and **MBPP**, selected for their diversity and real-world relevance. A fixed random seed was used to sample 50 tasks from each dataset, ensuring consistency. Each system was evaluated three times per dataset, and scores were averaged to reduce anomalies. No retries or result corrections were permitted, ensuring that every result reflected true system capability.

Evaluation Environment

All experiments were run on a local development machine (Asus Zenbook OLED 13, PopOS) with no dedicated GPU. While this limited throughput, it ensured a controlled environment free from network variability or cloud-side noise. Each run was executed sequentially, without batching or parallelism, allowing direct attribution of results to system design.

Evaluation Framework and Logging

A custom Python framework automated all evaluation steps—from task loading to result validation—removing human error and ensuring consistent execution. Logs captured key metrics such as taskID, execution time, agent interactions and task outcomes. This structured format ensured reproducibility and enabled fair downstream comparisons across setups. A sample log entry is shown below.

```
------------------------------------
Task ID: HumanEval/16
Execution Time: 12.67 seconds
Number of Agent Interactions: 5
Status: Passed
------------------------------------
```

Performance Assessment

System performance was evaluated along two key dimensions: **execution efficiency** and **solution reliability**. Execution time measured how quickly each setup processed tasks, reflecting not only processing speed but also architectural efficiency. MASON runs additionally captured overhead from inter-agent coordination, with longer times indicating increased complexity or latency.

Reliability was assessed using the *Pass@1* score, representing the rate of successful first-attempt completions without retries. This metric offered a clear measure of system accuracy under pressure.

To interpret results, each LLM was first tested individually to establish a performance baseline. These outcomes were then compared to MASON runs using the same models, isolating the impact of task specialization and collaboration. Further analysis across HumanEval and MBPP revealed how dataset complexity influenced performance under different architectures.

Together, these comparisons highlighted the trade-offs between coordination overhead and improved task distribution, offering a deeper understanding of how system design and LLM selection affect practical coding performance.

Results

Execution Time Analysis

Execution time measured how quickly each system processed tasks across different models, architectures, and datasets. Figure 3 and Table 6 show average execution times for MASON and single-agent setups.

MASON consistently introduced higher processing time across all models. The overhead was most pronounced in DeepSeek-Chat and Gemini 1.5 Flash, where MASON execution time more than doubled compared to single-agent runs. In contrast, GPT-4o Mini and Claude 3.5 Sonnet showed smaller timing differences, suggesting more efficient orchestration. MASON delays were also greater in the HumanEval (HE) dataset, likely due to its higher task complexity.

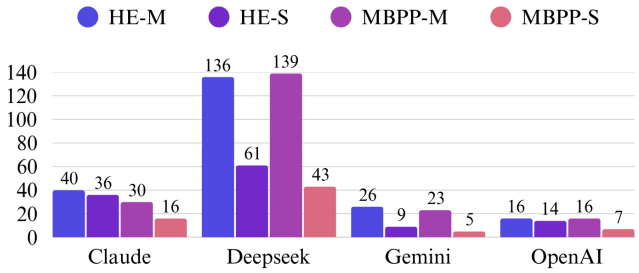

Figure 3: Execution Time by Model and Tasks

Model	HE-M	HE-S	MBPP-M	MBPP-S
Claude	39.92	36.18	30.72	15.84
DeepSeek	134.50	60.74	139.25	43.45
Gemini	25.59	8.79	23.35	4.81
OpenAI	15.65	13.54	15.94	6.51

Table 6: Average Execution Time Comparison

Pass Rate (Accuracy) Analysis

Accuracy scores measured how reliably each model produced correct outputs on the first attempt across datasets and system configurations. Figure 4 and Table 7 show average Pass@1 Scores for MASON and single-agent setups.

Claude 3.5 Sonnet achieved the highest and most stable accuracy overall, showing strong consistency in both MASON and single-agent modes. DeepSeek-Chat improved notably under MASON for MBPP tasks, suggesting benefits from task specialization in simpler scenarios. GPT-4o Mini showed minor gains with MASON, while Gemini 1.5 Flash underperformed in MASON on HumanEval tasks, likely due to difficulties with coordination and complex reasoning. Accuracy was generally more stable in MBPP, whereas HumanEval introduced greater variance due to higher task complexity.

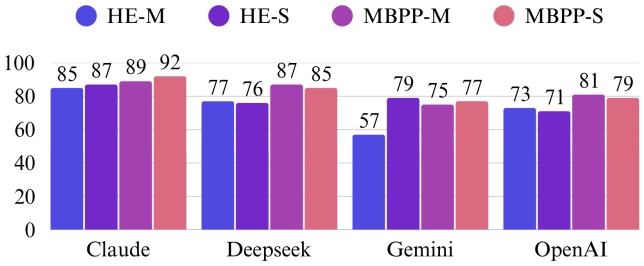

Figure 4: Pass Rate by Model and Tasks

Model	HE-M	HE-S	MBPP-M	MBPP-S
Claude	0.85	0.87	0.89	0.82
DeepSeek	0.77	0.76	0.87	0.85
Gemini	0.57	0.79	0.75	0.77
OpenAI	0.73	0.71	0.81	0.79

Table 7: Average Pass@1 Score Comparison

Failure Patterns and Error Density

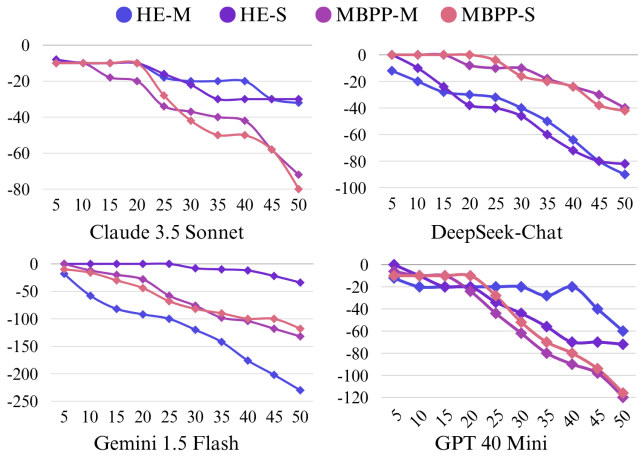

Figure 5: Cumulative Fail Dips over Tasks per Task

Failure trends from Figure 5 revealed how system design and task complexity influenced model robustness. Gemini 1.5 Flash showed sharp early failures under MASON, especially on HumanEval, suggesting difficulties with coordination and reasoning. MASON generally triggered earlier failure onset than single-agent systems, highlighting fragility during initial execution phases when inter-agent communication overhead is highest. Complex tasks in HumanEval led

to greater error density, while MBPP remained more stable across models.

As illustrated in Figure 6, HumanEval exhibited denser fail zones than MBPP, especially between Tasks 5–10, 25–30, and 35–40. These segments involved ambiguous instructions or edge cases. Gemini 1.5 Flash and DeepSeek-Chat struggled more broadly, while Claude 3.5 Sonnet and GPT-4o Mini showed localized weaknesses. The results suggest that task complexity—not just architecture—was a strong driver of failure.

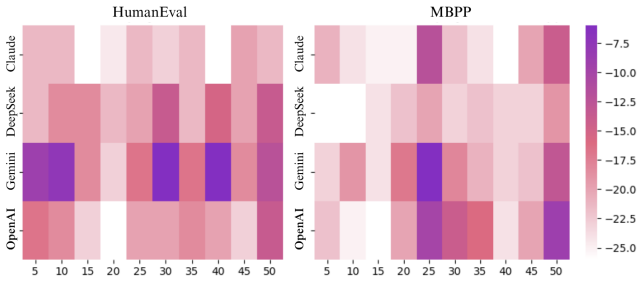

Figure 6: Fail Count over Tasks for both Datasets

Figure 7 highlights task-specific performance patterns. High-failure tasks such as `humaneval_Task1`, `humaneval_Task47`, and `mbpp_Task1` appeared across multiple models, revealing consistent difficulty. Conversely, MASON succeeded more on simpler MBPP tasks like `mbpp_Task26` and `mbpp_Task29`. These insights confirm that model stability is closely linked to both task complexity and coordination demands.

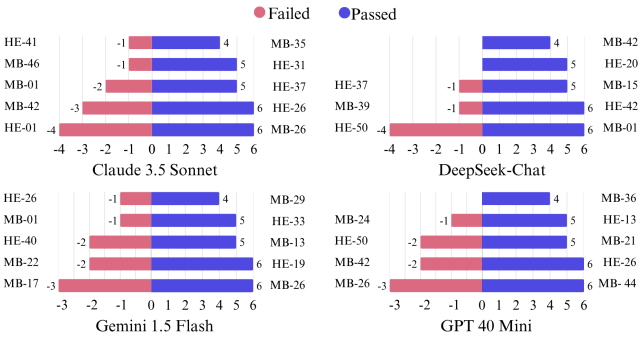

Figure 7: Top Passed and Failed Tasks per Model

Pass Rate Trend and Task Progression

Pass rate trends revealed how system performance evolved throughout the evaluation. As shown in Figure 8, MASON configurations—especially Gemini 1.5 Flash on HumanEval—exhibited early performance drops and struggled to recover, indicating coordination challenges under complex constraints. In contrast, single-agent setups like Claude 3.5 Sonnet and GPT-4o Mini maintained stable trajectories with fewer dips. DeepSeek-Chat MASON showed gradual recovery, suggesting adaptive coordination, while GPT-4o Mini MASON sustained high accuracy from the outset with minor fluctuations.

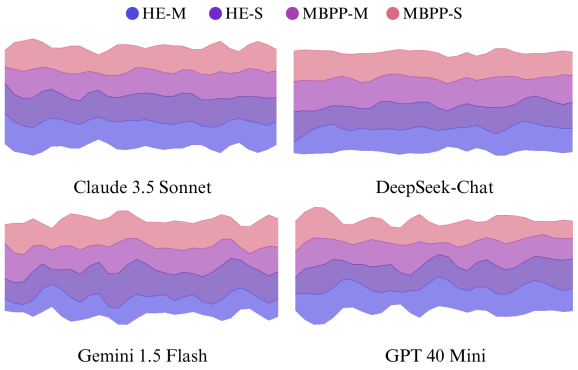

Figure 8: Pass Rate Trend over Tasks per Task

Cumulative scores in Figure 9 show that Claude 3.5 Sonnet and GPT-4o Mini MASON closed initial gaps and outperformed their single-agent counterparts in later tasks, demonstrating effective long-term orchestration. DeepSeek-Chat MASON lost early momentum, while Gemini 1.5 Flash continued to underperform—highlighting persistent weaknesses in managing extended sequences.

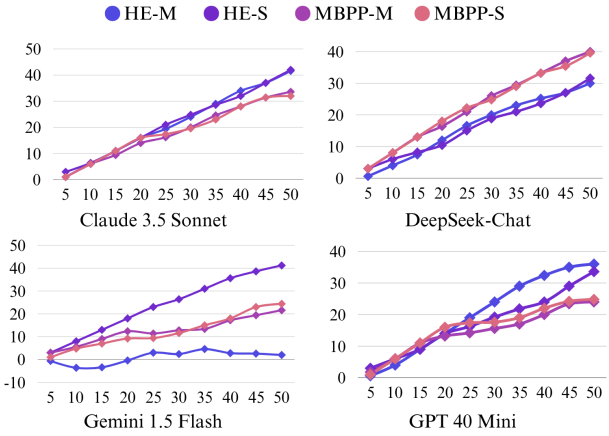

Figure 9: Cumulative Score over Tasks

Performance Summary and Model Ranking

To consolidate performance across multiple dimensions—accuracy, speed, and consistency—we analyzed radar charts per dataset, as shown in Fig 10 comparing MASON and single-agent setups. These visualizations illustrate model behavior under different system designs and task complexities, emphasizing the trade-offs in pursuing multi-agent collaboration.

Table 8 and Table 9 show that MASON offered marginal benefits in stability for Claude 3.5 Sonnet and GPT-4o Mini, though execution time remained comparable. DeepSeek-Chat gained accuracy under MASON in MBPP but saw minimal impact elsewhere. Gemini 1.5 Flash performed better as a single agent, with MASON impairing both accuracy and stability—particularly on complex HumanEval tasks. Here, PR refers to Pass Rate, RT to Runtime, and SB to Stability, helping summarize model performance differences across setups.

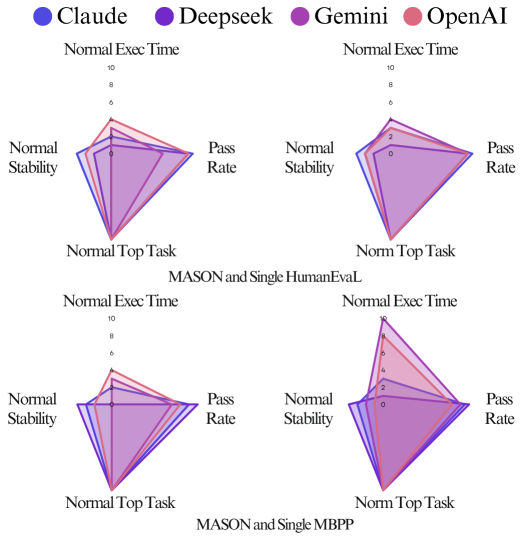

Figure 10: MASON vs Single LLM Model Ranking

Model	PR	RT	SB	Benefit
Claude 3.5 Sonnet	Yes	No	Yes	Stability
DeepSeek-Chat	Yes	No	Yes	Stability
Gemini 1.5 Flash	No	No	No	None
GPT-4o Mini	Yes	No	Yes	Stability

Table 8: HE — MASON vs Single-Agent Performance

Model	PR	RT	SB	Benefit
Claude 3.5 Sonnet	Yes	No	Yes	Balanced
DeepSeek-Chat	Yes	No	Yes	Stability
Gemini 1.5 Flash	No	No	No	None
GPT-4o Mini	Yes	No	Yes	Balanced

Table 9: MBPP — MASON vs Single-Agent Performance

Model	Strengths	Weaknesses
Claude 3.5 Sonnet	MASON and Single balanced and stable	Minor MASON gains
DeepSeek-Chat	Fastest execution, modest MASON stability gain	similar accuracy; MASON did not significantly help
Gemini 1.5 Flash	Fast in single-agent setup	MASON worsened accuracy and stability
GPT-4o Mini	Consistent, minor MASON edge in complex tasks	Slightly slower execution in MBPP

Table 10: Model Strength Summary

Evaluating Table 10 shows us that comparisons underscore a recurring theme: MASON introduced slight **stability gains** in some models but often at the cost of execution speed, with Gemini 1.5 Flash particularly affected by MASON overhead. Claude 3.5 Sonnet and GPT-4o Mini demonstrated the most balanced performance, benefiting from MASON only under specific conditions.

Conclusion

This study explored whether MASON using LLMs could enhance NCD by distributing tasks among specialized agents. Tested across several models and coding benchmarks, MASON was compared against conventional single-agent setups.

The results showed modest but consistent gains in stability and accuracy—most notably in simpler MBPP tasks. Claude 3.5 Sonnet and GPT-4o Mini performed reliably across both configurations, while DeepSeek-Chat benefited from MASON on select workflows. However, MASON came with a trade-off: increased execution time, and in the case of Gemini 1.5 Flash, reduced performance due to coordination overhead.

While MASON consistently outperformed single agents in stability, it fell short of delivering the broader performance gains initially expected. Key limitations stemmed from hardware and design: all evaluations ran on a CPU-only setup, restricting scale and forcing sequential execution. Open-source models—especially smaller ones—struggled under MASON due to high memory requirements and limited reasoning depth, making them unsuitable for deployment in low-resource environments.

Beyond hardware, the architecture itself limited performance. Shallow prompts and a rigid agent sequence likely prevented the system from fully leveraging collaborative reasoning. These challenges highlight the need for deeper prompt engineering, adaptive agent flows, and smarter orchestration—especially for complex, multi-step coding tasks.

Still, MASON successfully demonstrated its core capabilities: automated task delegation, agent interaction, and consistent performance tracking. With refinement, and adaptations for lightweight LLMs, this framework has real potential to support intelligent NCD—even in constrained environments.

Future Scope

There is strong potential to evolve this study into a production-ready tool. Future development should focus on improving orchestration through advanced prompt chaining, dynamic agent assignment, and deeper collaboration strategies. Expanding benchmark coverage and incorporating more diverse models would yield broader insights.

While this version used a lightweight YAML-based setup suitable for local use, scaling the system may benefit from adopting more robust agentic platforms such as LangChain, AutoGen, or a custom solution. A key goal is to replicate the study using only open-source models, enabling detailed architectural comparisons under resource constraints.

Future work should also explore MASON designs tailored for small LLMs—reducing orchestration complexity, simplifying task flows, or applying fine-tuning—to make agent-based systems viable in low-resource environments.

References

AlOmar, E. A.; Venkatakrishnan, A.; Mkaouer, M. W.; Newman, C.; and Ouni, A. 2024. How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations. In *Proceedings of the 21st International Conference on Mining Software Repositories*, 202–206.

ALSAADI, H. A.; RADAIN, D. T.; ALZAHRANI, M. M.; ALSHAMMARI, W. F.; ALAHMADI, D.; and FAKIEH, B. 2021. Factors that affect the utilization of low-code development platforms: survey study. *Romanian Journal of Information Technology & Automatic Control/Revista Română de Informatică și Automatică*, 31(3).

Aubry, M.; Meng, H.; Sugolov, A.; and Papyan, V. 2024. Transformer Alignment in Large Language Models. *arXiv preprint arXiv:2407.07810*.

Bahsi, E. M.; Ceyhan, E.; and Kosar, T. 2007. Conditional workflow management: A survey and analysis. *Scientific Programming*, 15(4): 283–297.

Balamurugan, A.; Shetty, H. A.; Sengunthar, K. M.; and Gupta, M. 2023. Auditing Low-Code and No-Code Platforms Securing Citizen Development. In *Modernizing Enterprise IT Audit Governance and Management Practices*, 68–94. IGI Global.

Bang, Y.; Cahyawijaya, S.; Lee, N.; Dai, W.; Su, D.; Wilie, B.; Lovenia, H.; Ji, Z.; Yu, T.; Chung, W.; et al. 2023. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023*.

Barbarroxa, R.; Gomes, L.; and Vale, Z. 2024. Benchmarking Large Language Models for Multi-agent Systems: A Comparative Analysis of AutoGen, CrewAI, and TaskWeaver. In *International Conference on Practical Applications of Agents and Multi-Agent Systems*, 39–48. Springer.

Beranic, T.; Rek, P.; and Hericko, M. 2020. Adoption and usability of low-code/no-code development tools. In *Central European Conference on Information and Intelligent Systems*, 97–103. Faculty of Organization and Informatics Varazdin.

Bock, A.; and Frank, U. 2021. Low-Code Platform. Business & Information Systems Engineering, 63 (6), 733–740.

Curty, S.; Härer, F.; and Fill, H.-G. 2022. Blockchain application development using model-driven engineering and low-code platforms: A survey. In *International Conference on Business Process Modeling, Development and Support*, 205–220. Springer.

Di Ruscio, D.; Kolovos, D.; de Lara, J.; Pierantonio, A.; Tisi, M.; and Wimmer, M. 2022. Low-code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling*, 21(2): 437–446.

Du, Y.; Li, S.; Torralba, A.; Tenenbaum, J. B.; and Mordatch, I. 2023. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*.

Duan, Z.; and Wang, J. 2024. Exploration of LLM Multi-Agent Application Implementation Based on LangGraph+ CrewAI. *arXiv preprint arXiv:2411.18241*.

Goonatilleke, S. T.; and Hettige, B. 2022. Past, Present and Future Trends in Multi-Agent System Technology. *Journal Européen des Systèmes Automatisés*, 55(6).

He, J.; Treude, C.; and Lo, D. 2024. LLM-Based Multi-Agent Systems for Software Engineering: Vision and the Road Ahead. *arXiv preprint arXiv:2404.04834*.

Hong, S.; Zheng, X.; Chen, J.; Cheng, Y.; Wang, J.; Zhang, C.; Wang, Z.; Yau, S. K. S.; Lin, Z.; Zhou, L.; et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Ishibashi, Y.; and Nishimura, Y. 2024. Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization. *arXiv preprint arXiv:2404.02183*.

Karamthulla, M. J.; Malaiyappan, J. N. A.; and Tillu, R. 2023. Optimizing Resource Allocation in Cloud Infrastructure through AI Automation: A Comparative Study. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, 2(2): 315–326.

Li, H.; Jiang, H.; Zhang, T.; Yu, Z.; Yin, A.; Cheng, H.; Fu, S.; Zhang, Y.; and He, W. 2023. TrainerAgent: Customizable and Efficient Model Training through LLM-Powered Multi-Agent System. *arXiv preprint arXiv:2311.06622*.

Liang, T.; He, Z.; Jiao, W.; Wang, X.; Wang, Y.; Wang, R.; Yang, Y.; Tu, Z.; and Shi, S. 2023a. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118*.

Liang, X.; Song, S.; Niu, S.; Li, Z.; Xiong, F.; Tang, B.; Wang, Y.; He, D.; Cheng, P.; Wang, Z.; et al. 2023b. Uhgeval: Benchmarking the hallucination of chinese large language models via unconstrained generation. *arXiv preprint arXiv:2311.15296*.

Lin, F.; Kim, D. J.; et al. 2024. When llm-based code generation meets the software development process. *arXiv preprint arXiv:2403.15852*.

Liu, L.; Yang, X.; Shen, Y.; Hu, B.; Zhang, Z.; Gu, J.; and Zhang, G. 2023a. Think-in-memory: Recalling and post-thinking enable llms with long-term memory. *arXiv preprint arXiv:2311.08719*.

Liu, P.; Yuan, W.; Fu, J.; Jiang, Z.; Hayashi, H.; and Neubig, G. 2023b. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9): 1–35.

Liu, Z.; Yao, W.; Zhang, J.; Yang, L.; Liu, Z.; Tan, J.; Choubey, P. K.; Lan, T.; Wu, J.; Wang, H.; et al. 2024. AgentLite: A Lightweight Library for Building and Advancing Task-Oriented LLM Agent System. *arXiv preprint arXiv:2402.15538*.

Logeshwaran, J.; Kiruthiga, T.; Kannadasan, R.; Vijayaraja, L.; Alqahtani, A.; Alqahtani, N.; and Alsulami, A. A. 2023. Smart load-based resource optimization model to enhance the performance of device-to-device communication in 5G-WPAN. *Electronics*, 12(8): 1821.

Luo, Y.; Liang, P.; Wang, C.; Shahin, M.; and Zhan, J. 2021. Characteristics and challenges of low-code development: the practitioners' perspective. In *Proceedings of the 15th*

ACM/IEEE international symposium on empirical software engineering and measurement (ESEM), 1–11.

Ma, C.; Zhang, J.; Zhu, Z.; Yang, C.; Yang, Y.; Jin, Y.; Lan, Z.; Kong, L.; and He, J. 2024a. AgentBoard: An Analytical Evaluation Board of Multi-turn LLM Agents. *arXiv preprint arXiv:2401.13178*.

Ma, L.; Yang, W.; Xu, B.; Jiang, S.; Fei, B.; Liang, J.; Zhou, M.; and Xiao, Y. 2024b. Knowlog: Knowledge enhanced pre-trained language model for log understanding. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 1–13.

Ma, Z.; Chen, A. R.; Kim, D. J.; Chen, T.-H.; and Wang, S. 2024c. Llmparser: An exploratory study on using large language models for log parsing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

Martinez, E.; and Pfister, L. 2023. Benefits and limitations of using low-code development to support digitalization in the construction industry. *Automation in Construction*, 152: 104909.

Miret, S.; and Krishnan, N. 2024. Are LLMs Ready for Real-World Materials Discovery? *arXiv preprint arXiv:2402.05200*.

Peng, J.; Wu, M.; Zhang, X.; Xie, Y.; Jiang, F.; and Liu, Y. 2006. A collaborative Multi-Agent model with knowledge-based communication for the RoboCupRescue simulation. In *International Symposium on Collaborative Technologies and Systems (CTS'06)*, 341–348. IEEE.

Perez, J.; Léger, C.; Ovando-Tellez, M.; Foulon, C.; Dussauld, J.; Oudeyer, P.-Y.; and Moulin-Frier, C. 2024. Cultural evolution in populations of Large Language Models. *arXiv preprint arXiv:2403.08882*.

Pinho, D.; Aguiar, A.; and Amaral, V. 2023. What about the usability in low-code platforms? A systematic literature review. *Journal of Computer Languages*, 74: 101185.

Rasheed, Z.; Sami, M. A.; Kemell, K.-K.; Waseem, M.; Saari, M.; Systä, K.; and Abrahamsson, P. 2024. CodePori: Large-Scale System for Autonomous Software Development Using Multi-Agent Technology. *arXiv preprint arXiv:2402.01411*.

Sahay, A.; Indamutsa, A.; Di Ruscio, D.; and Pierantonio, A. 2020. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 171–178. IEEE.

Sarumi, O. A.; and Heider, D. 2024. Large language models and their applications in bioinformatics. *Computational and Structural Biotechnology Journal*.

Schäfer, M.; Nadi, S.; Eghbali, A.; and Tip, F. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*.

Shabir, A.; Ahmed, K. T.; Kanwal, K.; Almas, A.; Raza, S.; Fatima, M.; and Abbas, T. 2024. A Systematic Review of Attention Models in Natural Language Processing. *STATISTICS, COMPUTING AND INTERDISCIPLINARY RESEARCH*, 6(1): 33–56.

Shen, C.; Yang, W.; Pan, M.; and Zhou, Y. 2023. Git Merge Conflict Resolution Leveraging Strategy Classification and LLM. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, 228–239. IEEE.

Shen, W.; Li, C.; Chen, H.; Yan, M.; Quan, X.; Chen, H.; Zhang, J.; and Huang, F. 2024. Small llms are weak tool learners: A multi-llm agent. *arXiv preprint arXiv:2401.07324*.

Solanky, J.; Patil, K.; and Patel, G. 2016. Resemblance of PhoneGap and Titanium for Mobile Application Development. *International Journal of Computer Applications*, 144(10).

Sufi, F. 2023. Algorithms in low-code-no-code for research applications: a practical review. *Algorithms*, 16(2): 108.

Talebirad, Y.; and Nadiri, A. 2023. Multi-agent collaboration: Harnessing the power of intelligent llm agents. *arXiv preprint arXiv:2306.03314*.

Topsakal, O.; and Akinci, T. C. 2023. Creating large language model applications utilizing langchain: A primer on developing llm apps fast. In *International Conference on Applied Engineering and Natural Sciences*, volume 1, 1050–1056.

Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.-A.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Wang, Z.; Chu, Z.; Doan, T. V.; Ni, S.; Yang, M.; and Zhang, W. 2024. History, development, and principles of large language models: an introductory survey. *AI and Ethics*, 1–17.

Wu, Q.; Bansal, G.; Zhang, J.; Wu, Y.; Li, B.; Zhu, E.; Jiang, L.; Zhang, X.; Zhang, S.; Liu, J.; et al. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*.

Xie, Z.; Chen, Y.; Zhi, C.; Deng, S.; and Yin, J. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764*.

Yan, Z. 2021. The impacts of low/no-code development on digital transformation and software development. *arXiv preprint arXiv:2112.14073*.

Yang, C.; Liu, J.; Xu, B.; Treude, C.; Lyu, Y.; Li, M.; and Lo, D. 2023. APIDocBooster: An Extract-Then-Abstract Framework Leveraging Large Language Models for Augmenting API Documentation. *arXiv preprint arXiv:2312.10934*.

Yuan, Z.; Lou, Y.; Liu, M.; Ding, S.; Wang, K.; Chen, Y.; and Peng, X. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*.

Zhang, Y.; Li, Y.; Cui, L.; Cai, D.; Liu, L.; Fu, T.; Huang, X.; Zhao, E.; Zhang, Y.; Chen, Y.; et al. 2023. Siren's song in the AI ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219*.