Yuma-Code: Automating LLM Agent Generation from Natural Language Specifications

Débora Souza
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil
debora.souza@ccc.ufcg.edu.br

Caíque Campelo
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil
caique.campelo@ccc.ufcg.edu.br

André Neves
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil
andre.neves@ccc.ufcg.edu.br

Patricia Machado
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil
patricia@computacao.ufcg.edu.br

Eliane Araújo
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil
eliane@computacao.ufcg.edu.br

Leandro Balby Marinho
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil
lbmarinho@computacao.ufcg.edu.br

Everton L.G. Alves
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil
everton@computacao.ufcg.edu.br

Abstract

Agentic systems are a new generation of software engineering applications that go far beyond traditional systems. Automatic generation of such systems is a promising approach to accelerate development, disseminate engineering knowledge, and improve construction scalability. Despite advances in LLM-based code generation, existing approaches provide limited support for structured requirement elicitation, architecture-aware generation, and systematic behavioral evaluation of agent-based systems. We propose Yuma-Code, a CLI framework that transforms natural-language specifications into executable agent code through structured requirement elicitation, automatic architecture selection, and multi-agent orchestration with specialized roles for requirements engineering, architecture design, and test generation. We evaluate Yuma-Code against four state-of-the-art frameworks using AgentGenBench, a benchmark encompassing four agent architectures (Simple, Tool, Memory, and MCP). Results show that Yuma-Code consistently achieves full specification compliance and generates evaluation artifacts via LLM-as-a-Judge across all scenarios. In more complex settings (MCP), it attains high clarity and relevance (0.97) and strong memory recall (0.85). Copilot CLI demonstrates competitive performance across configurations, while Gemini CLI achieves the strongest results in the Simple scenario. Spec-kit and Kiro frequently generated non-executable code. These results suggest that agent generation benefits from being treated as a structured software engineering process, and highlight behavioral validation as a critical yet underexplored dimension in current LLM-based code generation approaches.

Keywords

AI Agent Generation, LLM-Based Agents, Code Generation, Multi-Agent Systems, Specification-Driven Development, LLM-as-a-Judge

1 Introduction

Large Language Models (LLMs) have been increasingly explored in Software Engineering (SE) tasks [4, 14]. Evaluation efforts have ranged from function-level benchmarks such as HumanEval [7], MBPP [2], and APPS [12] to class-based and multi-component scenarios [10], while tools like GitHub Copilot[1] and Cursor[2] embed LLMs directly into development workflows.

When equipped with tools and reasoning capabilities, LLMs can act as *agents* capable of sequential decision-making and multi-step execution [25–27]. As interest in agentic systems grows, so does the demand for accessible ways to create them — from visual platforms such as n8n[3] and OpenAI Agent Builder[4] to code-oriented approaches like AutoAgent [23].

Visual platforms for agent development have lowered the barrier for non-developers by enabling the construction of workflows through graphical interfaces. However, code-based agents remain important for integration with existing software ecosystems, reproducibility, and systematic testing — properties critical in production and safety-sensitive contexts, and reflected in the growing adoption of frameworks such as LangGraph[5], which emphasize explicit control over execution flows and agent behavior.

Despite growing interest in agentic AI systems, the automatic end-to-end generation of code-based LLM agents from natural-language specifications remains limited. Generating an agent involves more than producing code — it requires constructing a coordinated system with tool usage, memory, and structured control flow [26], and building such systems demands substantial technical expertise [23] with no widely adopted way to specify agent behavior in a form directly translatable into executable and testable

[1] https://github.com/features/copilot
[2] https://cursor.com/
[3] https://n8n.io/
[4] https://platform.openai.com/agent-builder
[5] https://www.langchain.com/langgraph

implementations. Existing tools such as Spec-kit, Gemini CLI, and Copilot primarily target general-purpose code generation (e.g., APIs, scripts, or applications), offering limited support for agent-specific concerns: they lack systematic mechanisms for structuring specifications, deriving agent architectures, or validating agent behavior.

To address this gap, we introduce Yuma-Code (Your User-guided Multi-agent System for Code Generation), a command-line framework that translates natural-language specifications into executable agent code. Yuma-Code guides users through a structured specification process — collecting requirements such as functionalities and edge cases — and from this input automatically selects an agent architecture, integrates external tools or MCP connections, and produces executable and testable code.

The framework employs multi-agent orchestration, in which specialized sub-agents assume roles such as requirements engineer, software architect, and test designer, reducing the manual effort typically involved in code-based agent development. It also supports automated validation and a human-in-the-loop mode, enhancing maintainability, reliability, and traceability throughout the process [1, 8].

We evaluate Yuma-Code against four frameworks — Gemini CLI, Copilot CLI, Spec-kit, and Kiro — using AgentGenBench, a benchmark covering four agent architectures (Simple, Tool, Memory, MCP). Yuma-Code was the only framework to achieve full specification compliance (6/6) and LLM-as-a-Judge artifact generation across all scenarios, requiring zero manual corrections throughout — compared to 41 in the MCP scenario alone for Copilot CLI. In behavioral evaluation, Yuma-Code led in the MCP scenario (clarity and relevance of 0.97), while Gemini CLI achieved perfect scores in the Simple scenario (clarity, relevance, completeness = 1.0). In the Memory scenario, Spec-kit, Kiro, and Gemini produced memory recall below 0.23, while Copilot reached 0.68 — still below Yuma-Code's 0.85. In Tool and MCP scenarios, Spec-kit and Kiro frequently produced fragmented or non-executable code, with Kiro reaching 0/6 spec compliance in the MCP scenario.

In summary, this paper makes the following contributions:

(1) Yuma-Code, a multi-agent framework for end-to-end generation of code-based LLM agents from natural-language specifications;
(2) AgentGenBench, a benchmark spanning four agent architectures (Simple, Tool, Memory, and MCP) for evaluating agent generation frameworks;
(3) An empirical evaluation comparing Yuma-Code with existing code generation frameworks, demonstrating its effectiveness in generating executable and behaviorally correct agents.

The remainder of this paper is organized as follows: Section 2 reviews related work; Section 3 presents the Yuma-Code architecture; Sections 4 and 5 describe the evaluation methodology and results; Section 6 discusses findings; and Section 7 concludes the paper.

2 Related Work

Frameworks for Code Generation. LLM-based code generation has received significant attention, with models such as GPT-4 achieving strong results on established benchmarks [7, 20]. Several frameworks have emerged to support this task, including Gemini CLI[6], Claude Code[7], Spec-kit[8], Copilot CLI[9], and Kiro[10], which translate natural-language descriptions into executable artifacts such as functions, APIs, or command-line utilities. However, none supports guided specification elicitation or an integrated agent development pipeline — users must manually define architecture and validation logic, and automated testing or end-to-end evaluation remain largely absent.

Frameworks for Agent Generation. Beyond generic code generation, a growing ecosystem of tools aims to facilitate AI agent creation. Examples include n8n and Dify, that enable the composition of tool-based workflows through graphical interfaces, while XML-code-oriented solutions such as AutoAgent [23] and the OpenAI Agent Builder enable the composition of workflows through either visual interfaces or XML code. Despite these advances, no existing framework provides a fully automated pipeline that transforms natural-language requirements into executable agent python code, accompanied by automated testing and evaluation.

Agent Evaluation. Existing benchmarks target either code generation — such as HumanEval-X [28] and SWE-Bench [15], which focus on self-contained problems — or agent behavior, such as AgentBench [17], WebArena [29] and GAIA [21], which evaluate LLMs as agents across diverse tasks. However, none addresses the generation of agent code itself. Evaluating behavioral correctness further requires assessing semantic quality beyond output verification. The LLM-as-a-Judge [3, 18], offers scalable human-aligned assessments, though it introduces challenges related to non-determinism and prompt sensitivity. Yuma-Code addresses this by incorporating automated LLM-based evaluation into its validation phase, complemented by human-in-the-loop review.

3 Yuma-Code Architecture

The Yuma-Code architecture is guided by four complementary design principles: (1) *specification-guided development*, (2) the delegation of low-level code to external generation tools, (3) explicit *human-in-the-loop* elicitation and validation, and (4) a *dossier of agent archetypes* to guide implementation. Reflecting these principles, the architecture is organized into three main phases — Draft, Generation, and Validation — as illustrated in Figure 1.

Motivated by the demonstrated advantages of multi-agent architectures with specialized roles over single-agent approaches in SE tasks [22], Yuma-Code implements this pipeline through a network of nine specialized agents, as shown in Figure 2. Each agent assumes a distinct role: the *Requirements Engineer* conducts the elicitation dialogue; the *Architect* proposes and refines the architectural pattern; the *Code Generator*, *Structural Test Generator*, *Evals Generator*, and *E2E Eval Generator* produce the code, tests and evals artifacts; and the *Code Runner*, *Structural Test Runner*, and *Evals Runner* validate each artifact upon creation by executing. Agents communicate through a shared state graph, passing structured outputs — requirements documents, architecture proposals,

[6] https://github.com/google-gemini/gemini-cli
[7] https://www.claude.com/product/claude-code
[8] https://github.com/github/spec-kit
[9] https://github.com/github/copilot-cli
[10] https://kiro.dev/

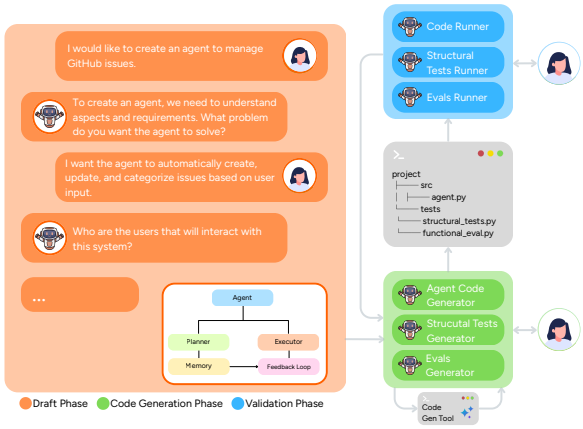

Figure 1: Overview of the Yuma-Code pipeline, illustrating its modular flow and agent interactions.

and share the environment with code and evaluation files — as inputs to downstream agents.

The nine-agent orchestration of Yuma-Code uses two models. The upstream agents (Requirements and Architect) are powered by GPT-4o (gpt-4o-latest), while the downstream agents (Code, Test, Evaluation Generators and the Runner agents) use Gemini 2.5 Flash through Gemini CLI.

Three explicit human-in-the-loop checkpoints structure the pipeline: (1) requirements and architecture approval, (2) evaluation criteria review , and (3) test case review. Failure handling differs by artifact type, reflecting the nature of each validation. Structural failures are unambiguous — a missing node or incorrect routing — and trigger automatic regeneration without human intervention. Functional evaluation (Evals) failures, however, may stem from incorrect agent logic, misconfigured criteria, or expected edge-case behavior; Yuma-Code therefore presents the user with an explicit choice: automatically refactor the agent, manually revise the test cases, or acknowledge and proceed.

To illustrate each phase concretely, we use a running example throughout this section: a user requesting an agent that manages **GitHub issues** using natural language — referred to as GitIssueBot.

3.1 Draft Phase: Capturing and Structuring Human Intent

The first phase of Yuma-Code is led by the Requirements Engineer Agent, which elicits user needs through a structured, interview-style dialogue (see the Draft Phase in Figure 1) rather than requiring a ready-made technical specification [1]. Drawing on requirements elicitation practices from SE, such as user stories [9], the agent progressively captures key dimensions — system purpose, end users, functionalities, and edge cases — along with acceptance scenarios explicitly elicited in Given/When/Then format, ensuring behavioral expectations are described concretely and in a testable manner. All dimensions must meet explicit completeness criteria, which define minimal information requirements for each dimension (e.g., presence of user roles, multiple functionalities, and at least one Given/When/Then scenario).

Running Example. To illustrate, consider a user who wants an agent to manage GitHub issues. The Requirements Engineer Agent initiates a structured dialogue:

The output of the Requirements Engineer Agent is a Requirements Document, reviewed and approved by the user, representing a portrait of user intent, which serves as the foundation for the subsequent phases.

Once requirements are elicited, the Architect Agent selects the most appropriate pattern from Yuma-Code's four built-in architectural definitions — *Simple*, *Memory-Augmented*, *Tool-Using*, and *MCP-Based* — which together cover the majority of agent architectures [16, 19, 24, 26]. In the *Simple* pattern, a single agent handles the entire workflow in a trigger–process–response loop with no memory or external interaction; in *Memory-Augmented*, the agent retains context across interactions to improve decision-making over time; in *Tool-Using*, the agent invokes external APIs or functions to complete tasks it cannot achieve alone; and in *MCP-Based*, interaction with external systems is abstracted through the Model Context Protocol. Reflection patterns are planned for future versions.

The architect agent reviews the requirements document and proposes the most suitable design — specifying the agent's name, role, main task, tool integrations, and a detailed operational flow covering input validation, processing steps, output generation, and edge case handling — which the user must review and approve, with the option to request modifications, before the pipeline advances to code generation.

Running Example. Based on the Requirements Document, the Architect Agent proposes the following architecture, presented to the user for approval:

3.2 Generation Phase: From Architecture to Executable Artifacts

The Code Generation Phase produces the agent code along with its associated tests and evaluations. Based on the validated architecture, Yuma-Code selects templates that capture the structural and behavioral patterns of the agent type and submits them, illustrated in Figure 4, together with architecture details, to an external code generation tool (e.g., Gemini CLI or Claude Code). These templates serve as contextual examples that guide the generation tool in producing correct, reproducible code — going beyond simple variable substitution toward a semantic construction of the agent guided by the tool's reasoning capabilities. Currently, the agent code is produced in LangGraph — the framework supported by Yuma-Code — but support for additional frameworks is planned in future work.

In addition to generating the agent code, Yuma-Code produces tests, functional evaluations and End2End evaluations. Structural tests verify architectural conformance, ensuring the graph has the correct nodes and routing, functional evaluations check whether the agent behaves according to specifications across nominal and edge cases, while end-to-end checks real integration dependencies for full execution. Functional and End2End evaluations use Deepeval[11] with an LLM acting as a judge to assess performance metrics such as task completion and, via GEval, criteria like clarity, completeness, and relevance. Before adoption, Yuma-Code allows the user to

[11] https://deepeval.com/

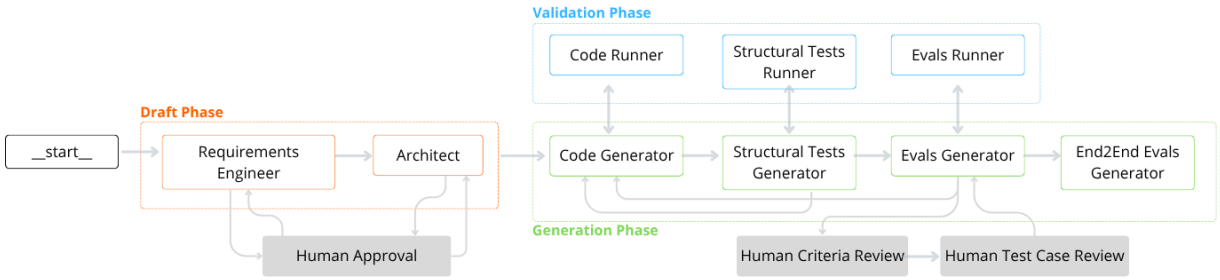

Figure 2: Yuma-Code's internal agent graph, comprising nine specialized agents and three human-in-the-loop checkpoints.

Running Example: Requirements Elicitation Dialogue

User: Create an agent to manage GitHub issues using natural language.
Agent: What problem do you expect to solve, and who are the primary users?
User: Allow developers to create, update, close, reopen, and list issues via natural language, ensuring consistent titles via MCP.
Agent: Describe a usage scenario in Given/When/Then format and list any edge cases.
User: *Given* the agent is connected via MCP; *When* the user requests an issue action specifying the repository; *Then* the agent executes it and returns a structured JSON. Edge cases: duplicates, missing permissions, vague titles, invalid repository.
...

Figure 3: Excerpt of the requirements elicitation dialogue.

Running Example: Architecture Proposal

Agent: GitHubIssueManagerAgent; **Type:** MCP-Based; **Task:** Manage GitHub issues via natural language (create, update, close, reopen, list); **Tool:** GitHub MCP.
Operational Flow: (1) Receive natural language command from the user; (2) validate input (clarity, repository specification, issue details); (3) execute the requested operation via MCP; (4) return structured JSON (issue ID, URL, status, feedback); and (5) on failure, report an error with an actionable message.

review each functional test criteria and generated test case, with the option to modify or add new ones.

Running Example. Upon architecture approval, Yuma-Code enters the generation phase:

Running Example: Code Generation Output

Status: ✓ Success **Details:** Python file generated successfully in the current directory.

Artifacts Created:
`git_issue_bot.py` — agent implementation
`test_agent_functional.py` — functional evaluations
`test_structural.py` — structural tests
`test_response.py` — end-to-end evaluations

The generation and validation phases interleave iteratively: each artifact is validated upon creation, and failures trigger targeted regeneration before the pipeline advances to the next artifact.

Example of code template for MCP agent

```
client = MultiServerMCPClient({"{mcp_name}": {"url": "{
    mcp_url}", ...}})
model = ChatOpenAI({model}).bind_tools(await client.
    get_tools())

async def {agent_name}(state: MessagesState):
    prompt = SystemMessage(content="""{prompt}""")
    response = await model.ainvoke([prompt] + state["
      messages"])
    return {"response": response}

async def build_graph():
    tool_node = ToolNode(await client.get_tools())
    builder = StateGraph(MessagesState)
    builder.add_node("agent", {agent_name})
    builder.add_node("tools", tool_node)
    ...
    return builder.compile()
```

Figure 4: Code template for MCP agent.

3.3 Validation Phase: Execution, Testing, and Evaluation

The Validation Phase ensures the correctness and functionality of the generated artifacts. Three agents oversee this process: the Code Runner checks the agent code for compilation and runtime issues, the Structural Tests Runner executes structural tests, and the Evals Runner verifies functional evaluations. Instead of running code directly, these agents delegate execution to the code generation tool, which installs dependencies, adapts inputs for non-interactive execution, runs the project, and reports results. Automated fixes, such as missing packages, are applied whenever possible.

The outcomes determine the next steps: successful validation advances the pipeline from structural testing to functional eval, while failures trigger regeneration of the faulty artifact (code or tests) before re-validation. This design ensures that generated agents are not only produced but also iteratively validated through execution and human oversight, guaranteeing robustness and fidelity to user intent.

Running Example. Yuma-Code enters the generation–validation loop, verifying each artifact before advancing to the next:

Running Example: Validation Output

Code Runner ✓ Success
Agent script executed successfully; dependencies resolved.

Structural Tests Runner ✓ Success
All 7 tests in `test_structural.py` passed, covering node behavior, graph routing, graph structure, and nominal and tool-based graph execution.

Evals Runner ✓ Success
Functional evaluations completed across 5 metrics. Clarity (0.89,), Relevance (0.94), Completeness (0.88), Argument Correctness (1.0) passed, and Tool Correctness scored 1.

4 Evaluation Setup

We designed an empirical study to evaluate the effectiveness and efficiency of Yuma-Code in comparison with other agentic frameworks, focusing on their ability to generate agent code, tests, and evaluation artifacts. This investigation is conducted from the perspective of researchers interested in assessing the completeness and reliability of the generation process. Following the Goal Question Metric (GQM) approach [5], the study is guided by the following research questions.

4.1 Research Questions

RQ$_1$ **How well do Yuma-Code and other frameworks generate valid and specification-compliant agent-based code?** To answer this question, the generated agents will be manually inspected to verify syntactic validity, specification alignment, and adherence to architectural constraints defined by the benchmark tasks.

RQ$_2$ **How well do Yuma-Code and other frameworks generate evaluation and test artifacts aligned with the agent specifications?** To address this question, the evaluation and test artifacts produced by each framework will be examined to determine whether they correctly reflect the expected evaluation logic and criteria of the original specifications.

RQ$_3$ **How well does the generated code perform in terms of executability and resource usage across agent types?** To address this question, the agent-related code artifacts produced by each framework will be executed to verify whether they can successfully run. Execution time and cost will be measured according to each framework's available instrumentation.

RQ$_4$ **How do the agents generated by each framework perform when executed against the ground-truth evals?** This question will be investigated by executing each generated agent on a set of evaluation scripts and analyzing quantitative outcomes such as success rates, as well as qualitative aspects of behavior.

RQ$_5$ **What is the overall quality of the code generated by Yuma-Code and other frameworks?** To answer this question, static code analysis tools will be applied to assess the maintainability, readability, and structural complexity of the generated agent implementations.

We developed the AgentGenBench benchmark [6, 30] as a structured evaluation setup composed of representative tasks, evaluation metrics, and the supporting infrastructure required to investigate these research questions in a systematic and reproducible way.

Table 1: Agents used to evaluate agent generation.

Agent Type	Agent Goal
Simple	**ENEM Essay Evaluator:** Evaluate essays across the five INEP competencies, producing per-competency scores, a total score, and detailed feedback.
Tool	**Weather Assistent:** Interpret user intent (current, forecast, trend) to query an external weather API.
Memory	**Tourist Assistant:** Provide structured information about cities and tourist attractions through persistent context and retrieval of previously stored knowledge.
MCP	**Git Issue Bot:** Manage GitHub issues via natural language (create, update, close, reopen, list) through MCP.

4.2 Tasks for Evaluation of Agent Generation

To evaluate agent generation capabilities, we define AgentGenBench, a benchmark comprising four tasks that represent distinct architectural patterns commonly found in agent-based systems: Simple (a direct LLM call with no additional components), Memory (an agent that maintains conversational context across turns), Tool (an agent that invokes an external weather API), and MCP (an agent that integrates an external system via the Model Context Protocol, a standard for sharing context between LLMs and external services). Together, these tasks cover four different agent architectures, ensuring that the benchmark captures a representative range of agent behaviors rather than a single interaction pattern. For each task, AgentGenBench provides: (i) a natural-language specification, (ii) a ground-truth implementation used as an oracle, (iii) functional evaluations using LLM-as-a-judge, and (iv) a contextual prompt to guide generation and execution. Table 1 illustrates a concrete example of each task. Full specifications are available in the replication package at Section 7.

4.3 Baseline Frameworks

The frameworks presented in Table 2 were selected for comparison due to their shared characteristics with Yuma-Code, including support for specification-driven code generation and AI-assisted workflows. To our knowledge, no existing framework is specifically designed to generate code-based agentic systems from natural-language specifications — frameworks such as MetaGPT [13] and AutoGen [11] automate software development, but target traditional applications, rather than agents. General-purpose coding agents such as Claude Code [12] were considered but excluded due to cost constraints. The selected frameworks thus provide a strong baseline for assessing how general-purpose systems perform when tasked with generating agent-based applications from specifications.

4.4 Experiment Execution Setup

We executed each framework across the four agent types defined in AgentGenBench, providing in each case (i) a natural-language specification and (ii) a contextual prompt defining key concepts and requirements (e.g., agent definition, use of tools or MCP, and acceptance criteria). All executions were conducted manually through

[12] https://claude.ai/

Table 2: Comparative feature analysis of agent generation frameworks. Spec Guidance: guided specification elicitation; QA: code validation and execution; Dev Workflow: structured development pipeline; Multi-Agent: multi-agent architecture; Agent Eval: LLM-as-a-Judge validation.

Framework	Domain	Open Source	Spec Guidance	QA	Dev Workflow	Multi-Agent	Agent Eval
Spec-kit	General	✓	✗	✗	✓	✗	✗
Gemini CLI	General	✓	✗	✗	✗	✗	✗
Copilot CLI	General	✓	✗	✓	✗	✗	✗
Kiro (IDE)	General	✗	✗	✓	✗	✓	✗
Yuma-Code	**Agent**	✓	✓	✓	✓	✓	✓

each framework's intended interface, with a human user communicating with the system when needed. Each framework was executed once per agent type to generate the corresponding artifacts (agent code, tests, and evaluations), with sessions isolated to avoid cross-task contamination by resetting the context window. For agents requiring external integrations, minimal setup was provided (e.g., API access or MCP endpoints). To account for non-determinism in agent behavior, each generated agent was executed three times against the ground-truth evaluations when answering RQ4, with the final score computed as the average across the three runs. All generated artifacts, execution logs, and evaluation results are publicly available at Section 7.

4.5 Framework Assessment Methodology

We evaluate the generated agents across four dimensions: code generation quality, assessed through specification compliance, integration with external components (e.g., tools, memory, or MCP), and structural completeness; test and evaluation artifact generation, capturing whether frameworks produce supporting artifacts such as structural tests or LLM-based evaluation scripts; executability and resource usage, measuring whether the produced code runs without manual corrections; and behavioral correctness, assessed by executing generated agents against predefined test cases. Results are classified using a three-level scale — *Total* (✓), *Partial* (✦), and *Misaligned* (✗) — with criteria defined per metric as described in each RQ.

5 Evaluation

This section presents the results of our target techniques and addresses the research questions. Following the evaluation setup described in Section 4, each task defined in AgentGenBench (Section 4.2) was provided as input to all frameworks, which then produced corresponding agent implementations subsequently assessed across the dimensions defined in Section 4.5.

5.1 RQ_1: Code Generation Analysis

Table 3 summarizes performance across nine dimensions, with all evaluations manually verified. For code generation quality, we focus on three metrics — *Specification Compliance*, *Layer Integration*, and *Agent Code Generation* — each classified as *Total* (✓), *Partial* (✦), or *Misaligned* (✗), with criteria defined per metric. [13]

In the Simple scenario, Gemini CLI, Copilot, and Yuma-Code consistently achieved full specification compliance and agent code generation. In the Tool scenario, however, Gemini CLI partially succeeded, producing code without prompts, and Kiro only partially integrated the tool layer, while Spec-kit relied on mocked components in both cases. Notably, Kiro's approach of mocking the LLM even in the Simple scenario highlights fundamental usability and integration limitations.

In the Memory agent scenario, Yuma-Code was the only framework successfully implementing all layers, including memory, achieving full compliance and producing fully functional agent code. Gemini CLI and Copilot generated partial code, but Gemini lacked memory implementation, and Copilot implemented memory incorrectly. Kiro and Spec-kit showed incomplete or mocked memory handling, further illustrating difficulties in layering multiple agent components correctly.

The MCP scenario, which involves remote GitHub access, revealed more pronounced differences. Only Yuma-Code and Copilot CLI produced fully compliant agents with proper MCP integration, handling edge cases and implementing complete LangGraph + LLM + CLI layers. Gemini CLI generated functional agent code but failed to integrate MCP correctly, relying instead on direct API calls. Spec-kit and Kiro were critically misaligned: Spec-kit generated mere placeholders, while Kiro produced only a CLI skeleton without any LLM or MCP integration. Figure 5 contrast both extremes: Yuma-Code connecting via `MultiServerMCPClient` using the standardized transport protocol, and Spec-kit returning hardcoded URLs — with a comment in the generated code itself acknowledging that a real implementation is still needed.

Overall, the results indicate a clear hierarchy in agent generation capabilities. Yuma-Code and Copilot CLI consistently stand out across all agent types, producing complete, executable code with correct integration of required layers. Gemini CLI performs adequately in simpler configurations but struggles with remote MCP and memory integration. Spec-kit and Kiro generally underperform, often generating partial code or relying on mocks, which limits practical usability. These findings highlight that the ability to correctly integrate all required layers is a stronger predictor of agent functionality than mere specification compliance.

An explanation for the performance gap is that general-purpose code generation frameworks are not specifically designed for agent generation, and thus may lack built-in awareness of LangGraph conventions, MCP protocols, or agent-specific patterns. To mitigate this threat to validity, all frameworks received a context prompt providing key definitions ensuring that domain knowledge was equally available to all.

5.2 RQ_2: Test and Evaluation Generation

To address RQ_2, we examine the Test Generation and LLM-as-a-judge Eval (Agent Evaluation) dimensions from Table 3. Each one was classified as *Total* (✓), *Partial* (✦), or *Misaligned* (✗), with criteria defined per metric. [14] These dimensions capture whether each

[13] *Specification Compliance* scores six equally weighted criteria (agent workflow, LLM node, integration layer, core actions, structured output, and edge case handling): ✓ ≥5, ✦ 2–4, ✗ <2. *Layer Integration* evaluates connection to real external layers (MCP, memory, or tools): ✓ real integration, ✦ incorrect implementation, ✗ absent. *Agent Code Generation* assesses full component presence (workflow, LLM node, integration layer, CLI): ✓ fully implemented, ✦ mocked or misconfigured, ✗ absent.

[14] Test generation classification: ✓ (*Total*) indicates that the framework generates tests specifically for the agent (unit, structural, or behavioral, with or without mocks); ✦ (*Partial*) indicates that tests are generated only for auxiliary components (e.g., helper

Table 3: Comparative evaluation of all agent variants across frameworks

	ENEM Essay Evaluator (Simple)					Tourist Assistant (Memory)				
	Gemini	Copilot	Spec-kit	Kiro	YUMA-Code	Gemini	Copilot	Spec-kit	Kiro	YUMA-Code
Spec Compliance	✓ 6/6	✓ 6/6	✦ 4/6	✦ 3/6	✓ 6/6	✓ 5/6	✓ 6/6	✓ 5/6	✦ 3/6	✓ 6/6
Agent Code Gen.	✓	✓	✦	✦	✓	✦	✓	✦	✦	✓
Layer Integration	-	-	-	-	-	✗	✦	✗	✗	✓
Test Generation	✦	✦	✦	✦	✓	✗	✓	✗	✓	✓
LLM-as-a-Judge Eval Gen.	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓
Code Executability	✓	✓	✗	✓	✓	✓	✗	✗	✓	✓
Exec Time (min)	10.9	4.1	28.3	168.9	8.0	10.5	4.4	28.2	168.5	9.0
Cost ($)	1	–	0.43	-	0.04	0.76	–	0.72	–	0.05
Files count	7	8	22	63	7	4	5	14	31	5
	Weather Assistant (Tool)					**GitIssueBot (MCP)**				
	Gemini	Copilot	Spec-kit	Kiro	YUMA-Code	Gemini	Copilot	Spec-kit	Kiro	YUMA-Code
Spec Compliance	✦ 3/6	✓ 6/6	✗ 1/6	✓ 5/6	✓ 6/6	✓ 5/6	✓ 6/6	✗ 1/6	✗ 0/6	✓ 6/6
Agent Code Gen.	✦	✓	✗	✦	✓	✦	✓	✗	✗	✓
Layer Integration	✗	✓	✗	✓	✓	✗	✓	✗	✗	✓
Test Generation	✓	✓	✓	✓	✓	✗	✓	✦	✦	✓
LLM-as-a-Judge Eval Gen.	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓
Code Executability	✗	✓	✓	✓	✓	✗	✓	✗	✗	✓
Exec Time (min)	43.3	31.3	35.1	102.0	69.4	12.5	20	21.3	55	8.1
Cost ($)	0.4	-	0.6	-	0.05	0.5	-	3.81	-	0.05
Files count	8	15	18	53	7	7	9	22	30	7

YUMA-CODE: real MCP integration

```python
client = MultiServerMCPClient({
  "github": {
    "url": "https://api.githubcopilot.com/mcp/",
    "headers": {"Authorization":
      f"Bearer {os.getenv('GITHUB_TOKEN')}"},
    "transport": "streamable_http",
  }
})
async def get_tools():
    tools = await client.get_tools()
    return tools
```

Spec-kit: mocked MCP integration

```python
def create_issue(self, repository: str,
                 title: str,
                 description: str = None):
  # This is a simplified example.
  # A real implementation would use
  # the MCP client to make a call
  # to the GitHub API.
  print(f"Creating issue:'{title}'...")
  return f"https://github.com/{repository}/issues/1"
```

Figure 5: Comparison between YUMA-CODE and Spec-kit MCP integration.

framework produces test and evaluation artifacts alongside the agent code. No framework was instructed to generate tests.

Across all scenarios, YUMA-CODE was the only framework consistently generating both structural and functional test artifacts, including LLM-as-a-Judge evaluation. Competing frameworks — Gemini CLI, Copilot, Spec-kit, and Kiro — produced at most structural or black-box tests relying on mocked LLMs, sufficient for unit-level checks but unable to capture real agent behavior. Notably, Gemini CLI generated no test artifacts in the MCP scenario, and Spec-kit's unit tests became deprecated after the latest iteration in the Simple scenario.

YUMA-CODE uniquely validated both functional and structural aspects across all agent types, ensuring semantic alignment with the specification via LLM-as-a-Judge. Figure 6 illustrates the contrast between the artifact types generated in the MCP scenario: Copilot CLI produces structural unit tests without addressing agent behaviour, while YUMA-CODE generates LLM-as-a-Judge test cases with semantic criteria and expected tool calls.

Overall, YUMA-CODE emerges as the only framework capable of producing fully automated, functional, and structural test artifacts across all agent types, including LLM-as-a-Judge evaluation. Copilot and Kiro show moderate capabilities, generating some black-box or structural tests but lacking LLM-driven assessment. Gemini CLI and Spec-kit frequently fail to deliver meaningful evaluation artifacts, particularly in scenarios requiring functional validation or integration testing. These results emphasize that automated evaluation using LLM-as-a-Judge is a decisive differentiator in assessing agent quality and readiness.

5.3 RQ$_3$: Executability and Resource Usage

The final metrics from Table 3 cover Code Executability, Execution Time, Cost, and Files Count. In the Simple scenario, Gemini CLI, Copilot CLI, Kiro, and YUMA-CODE produced executable agents, while Spec-kit failed due to mocked LLM calls and import errors. Execution times varied widely, with YUMA-CODE and Copilot CLI the fastest and Kiro exceeding 1h40m. YUMA-CODE and Gemini CLI produced concise structures (7 files each), while Spec-kit and Kiro generated significantly larger codebases (22 and 63 files).

In the Memory scenario, executability issues became more evident: Copilot CLI and Spec-kit failed due to environment and state

modules or services), not for the agent itself; ✗ (*Misaligned*) indicates that no tests are generated.

Copilot CLI: structural test

```python
def test_repository_validation(self):
    from gitissuebot import GitIssueBot
    self.assertTrue(hasattr(GitIssueBot, '
      _validate_repository_format'))
    self.assertTrue(hasattr(GitIssueBot, '
      _validate_title_length'))
```

Yuma-Code: LLM-as-a-Judge test case

```python
LLMTestCase(
  input="Create issue 'Bug Report' in repo 'org/repo'.",
  expected_output='{"status":"success", "repo":"org/repo",
      "title":"Bug Report"}',
  expected_tools=[ToolCall(
    name="create_github_issue",
    input_parameters={"repo": "org/repo", "title":"Bug
      Report"})],
  metrics=[Clarity, Relevance, Completeness,
      ToolCorrectness, ArgumentCorrectness]  # GEval
)
```

Figure 6: Comparison between Copilot CLI structural testing and Yuma-Code LLM-as-a-Judge evaluation.

management problems, while Gemini CLI, Kiro, and Yuma-Code succeeded, with execution times ranging from 4 minutes to nearly 3 hours and Yuma-Code maintaining a compact structure (5 files versus Kiro's 31). In the Tool scenario, Yuma-Code and Copilot CLI remained executable and efficient, while Gemini CLI failed, and Kiro and Spec-kit produced larger but functional artifacts, following similar trends in time and cost.

The MCP scenario posed the greatest challenge, with only Yuma-Code and Copilot CLI generating executable agents, while the remaining frameworks failed due to incomplete implementations or integration issues. Yuma-Code achieved the best performance in both execution time (8m 19s) and cost efficiency ($0.03, excluding undisclosed components), while Kiro required substantially more time and resources.

A key factor behind these differences is how each framework structures the generation process. Spec-kit and Kiro adopt a granular, step-by-step decomposition with iterative refinements, which increases execution time. In contrast, Yuma-Code follows a more guided and integrated strategy, structuring the specification and generating the agent based on templates, which contributes to its executability, lower execution time, and more compact artifacts. Copilot CLI, although not specification-driven, also benefits from a more direct generation approach, explaining its strong performance in efficiency and executability.

5.4 RQ_4: Agents under LLM-as-a-Judge Eval

To address **RQ_4:**, only minimally functional agents were included, with adjustments limited to import corrections, structured output handling, and basic configuration alignment; agents requiring broader fixes were excluded. The results are summarized in Table 4.

All included agents were evaluated three times using LLM-as-a-judge, scoring G-Eval metrics provided by the DeepEval framework[15]. The final score for each metric is the average across the three runs.

In the Simple scenario, Gemini CLI, Copilot CLI, and Yuma-Code required no code modifications, while Kiro needed minor adjustments and Spec-kit demanded extensive edits to replace mocks, restore LLM calls, and fix JSON parsing. Gemini achieved perfect scores (1.0), followed by Yuma-Code (0.95, 0.99, 0.95) and Copilot CLI (0.79, 0.96, 0.95), while Spec-kit (0.72, 0.86, 0.87) and Kiro (0.62, 0.57, 0.37) showed lower performance, reflecting structural fragility and semantic incompleteness.

In the Memory scenario, only Yuma-Code and Copilot CLI were functional without rework, while Gemini and Spec-kit required minor fixes and Kiro needed substantial adjustments. Results were heterogeneous: Yuma-Code showed strong memory recall (0.85) but low clarity (0.13) due to compact outputs; Gemini achieved high clarity (0.80) and relevance (0.94) but poor memory recall (0.20); Copilot CLI balanced relevance (0.82) with moderate recall (0.68). Spec-kit performed moderately (clarity 0.56, recall 0.73), while Kiro degraded across metrics (clarity 0.36, recall 0.28), reflecting difficulties in maintaining state. These results highlight the importance of architecture-aware metrics beyond purely semantic criteria.

In the Tool scenario, Gemini and Spec-kit were excluded due to external dependency issues. Among the remaining frameworks, all achieved perfect argument correctness (1.0), while tool correctness varied (Copilot 0.4, Kiro 0.8, and Yuma-Code 0.4). Kiro and Copilot showed higher clarity (1.0 and 0.95), whereas Yuma-Code produced more concise outputs (clarity 0.41) but led in relevance (0.93) and task completion (0.63), suggesting more effective end-to-end execution. The MCP scenario was more complex, requiring OAuth flow and GitHub API handling. Despite this increased complexity, Yuma-Code again achieved the strongest overall performance, with clarity (0.97), relevance (0.97), completeness (0.95), and perfect argument correctness (1.0). Gemini CLI maintained good clarity (0.85) and tool correctness (0.90), but its lower relevance (0.57) and task completion (0.52) indicate limited contextual integration. Copilot CLI achieved moderate clarity (0.83) but weak tool correctness (0.30), often failing to align tool arguments properly.

Overall, Yuma-Code demonstrates the most consistent performance across all agent types, requiring no manual code changes in any scenario. Copilot CLI is the closest competitor, excelling in Tool and MCP scenarios and leading in memory consistency. Gemini CLI achieves high semantic precision in simple tasks but degrades under complexity. Kiro and Spec-kit show competitive isolated metrics but suffer from higher variability and frequent exclusions due to executability issues.

5.5 RQ_5: Code Quality

Beyond functional correctness, we assessed the structural quality of all generated artifacts—including agent code, tests, and evaluation

[15] Metrics: **Clarity** (well-structured, unambiguous output), **Relevance** (addresses the user's request appropriately), **Completeness** (all required components present), and **task-specific** metrics such as memory recall, tool correctness, and task completion. Each produces a score in [0,1].

Table 4: LLM-as-a-Judge evaluation of generated agents across frameworks

Metric (mean)	ENEM Essay Evaluator (Simple)					Tourist Assistant (Memory)				
	Gemini	Copilot	Spec-kit	Kiro	YUMA	Gemini	Copilot	Spec-kit	Kiro	YUMA
Clarity	1.00	0.79	0.72	0.58	0.95	0.80	0.41	0.56	0.36	0.13
Relevance	1.00	0.96	0.86	0.63	0.98	0.94	0.82	0.73	0.56	0.61
Completeness	1.00	0.95	0.87	0.35	0.94	0.85	0.83	0.79	0.38	0.36
Memory Recall	–	–	–	–	–	0.20	0.68	0.17	0.23	0.85
#Changed Lines	0	0	20	3	0	2	3	3	0	0
Metric (mean)	**Weather Assistant (Tool)**					**GitIssueBot (MCP)**				
	Gemini	Copilot	Spec-Kit	Kiro	YUMA	Gemini	Copilot	Spec-Kit	Kiro	YUMA
Clarity	–	0.95	–	1	0.41	0.85	0.83	–	–	0.97
Relevance	–	0.76	–	0.83	0.93	0.57	0.76	–	–	0.97
Completeness	–	0.78	–	0.84	0.32	0.54	0.70	–	–	0.95
Task Completion	–	0.60	–	0.49	0.63	0.52	0.46	–	–	0.65
Tool Correctness	–	0.4	–	0.8	0.4	0.90	0.30	–	–	0.80
Argument Correctness	–	1.00	–	1.00	1.00	0.90	0.88	–	–	1.00
#Changed Lines	–	0	–	0	0	11	41	–	–	0

Table 5: Comparative lint evaluation of generated agents across frameworks

Metric	ENEM Essay Evaluator (Simple)					Tourist Assistant (Memory)				
	Gemini	Copilot	Spec-kit	Kiro	YUMA	Gemini	Copilot	Spec-kit	Kiro	YUMA
Pylint	7.15/10	7.02/10	4.23/10	4.97/10	3.72/10	8.18/10	5.71/10	5.61/10	5.36/10	6.75/10
Flake8 (count)	52	83	63	4532	77	21	47	67	2122	61
Radon (cyclomatic)	3.556	3.083	4.0	3.8	3.3	2.3	2.4	2	4.0	2.6
Radon (maintainability)	87.0	47.9	76.4	56.6	71.9	72.4	60.7	64.1	61.2	69.8
Metric	**Weather Assistant (Tool)**					**GitIssueBot (MCP)**				
	Gemini	Copilot	Spec-kit	Kiro	YUMA	Gemini	Copilot	Spec-kit	Kiro	YUMA
Pylint	6.36/10	6.23/10	6.82/10	5.49/10	5.88/10	7.52/10	6.00/10	2.88/10	5.16/10	6.82/10
Flake8 (count)	12	244	25	2404	111	102	48	47	988	88
Radon (cyclomatic)	1.5	4.1	2.8	3.9	2.73	2.2	3.4	2.2	3.0	2.9
Radon (maintainability)	82.9	54.4	84.49	66.7	68.7	87.0	53.1	93.4	66.3	90.2

scripts—using Pylint[16] (quality score, 0–10), Flake8[17] (style violations), and Radon[18] (cyclomatic complexity and maintainability index, 0–100).

Table 5 reveals heterogeneous code quality across frameworks and scenarios. Gemini CLI consistently achieves the highest Pylint scores and maintainability, while Yuma-Code performs competitively, particularly in MCP (MI = 90.2). Cyclomatic complexity remains moderate across all frameworks (CC = 1.5–4.0), indicating generally simple control flows. The main differentiator is stylistic conformity, with Flake8 violation counts varying by orders of magnitude. Kiro illustrates this trend most clearly: despite acceptable Pylint scores and complexity, it accumulates 4,532 style violations in the Simple scenario and 2,122 in Memory, indicating structurally functional but stylistically inconsistent code. Spec-kit exhibits a similar pattern in the Tool scenario (2,404 violations) and, in MCP, combines the lowest Pylint score (2.88/10) with the highest maintainability index (93.4), highlighting that these metrics capture distinct, not always correlated, aspects of code quality.

[16] https://pylint.readthedocs.io/
[17] https://flake8.pycqa.org/
[18] https://radon.readthedocs.io/

Table 6: Flake8 violation categories in GitIssueBot (MCP).

Category	Gemini	Copilot	Spec-kit	Kiro	YUMA
E (Style)	43	90	42	897	83
W (Warnings)	6	243	0	710	1
F (Pyflakes)	2	16	5	35	4
Total	48	343	47	988	88

Table 6 summarizes Flake8 violations by category in the MCP scenario. Kiro and Copilot mainly produce W-category warnings (predominantly W293), which are cosmetic, whereas Gemini and Yuma-Code are dominated by E-category violations (primarily E501 and E302), reflecting minor readability issues. Spec-kit differs by exhibiting 27.7% E402 errors (imports outside module scope), a structurally riskier pattern that may lead to runtime failures. Overall, violations are predominantly cosmetic or stylistic across all frameworks.

6 Discussion

These results reveal trade-offs across frameworks, discussed below. **Implications for Developers.** Framework selection involves

distinct trade-offs. Yuma-Code is best suited for development workflows requiring specification guidance, automatic architecture selection, and integrated generation of tests and evaluation artifacts, enabling traceability and behavioral validation. Copilot CLI offers a practical alternative for rapidly producing well-structured agents with minimal manual intervention. Gemini CLI performs well for simple, single-turn agents but degrades in more complex integrations. Although Kiro and Spec-kit provide structured generation workflows, their reliance on mocked components and added complexity currently limit their practical utility.

Implications for Researchers. Existing benchmarks either target code generation at function or class level (e.g., HumanEval [7], ClassEval [10]) or evaluate agent behavior and task completion (e.g., GAIA [21], AgentBench [17], WebArena [29]) — but none addresses the generation of agent code itself. To our knowledge, AgentGenBench is the first benchmark for evaluating agent generation across architectural variants, encompassing specification compliance, behavioral correctness, and LLM-as-a-Judge artifact generation. The widespread failures in tool integration, MCP connectivity, and memory handling indicate that agent-specific generation remains underexplored and highlight AgentGenBench as a reusable benchmark for future research.

Implications for Tool Builders. The comparative results point to design characteristics that correlate with stronger agent generation outcomes. Frameworks that guide users through structured specification elicitation produce more complete and consistent inputs for generation. Architecture-aware generation using predefined templates reduces hallucinations and improves structural conformance. Moreover, generating tests and LLM-as-a-Judge scripts alongside agent code enables behavioral validation beyond syntactic analysis. While these patterns were consistent across the evaluated frameworks, ablation studies are needed to establish causal links between specific design choices and outcomes. **Threats to Validity - External validity.** The findings may not generalize beyond the four agent architectures and five frameworks evaluated. Each tool relies on distinct underlying LLMs and runtime environments, and model updates or configuration differences may alter future results. Additionally, AgentGenBench was designed with Yuma-Code's capabilities in mind, which may introduce benchmark bias. This threat is partially mitigated by the competitive performance of Copilot CLI — a general-purpose framework — across all scenarios, and by the context prompt with agents' related terms definitions provided to all frameworks, which equalized domain knowledge availability. **Internal validity.** Experiments required minimal manual intervention to correct imports and align configurations, which could introduce unintentional biases. In order to mitigate this, we standardized, these adjustments. Besides that, the limited number of agents per framework also constrains statistical robustness. To mitigate generation variability, we implemented measures on the inputs. We made an extra effort to remove ambiguities and omissions from the natural-language specifications to minimize the need for further clarification by providing contextual prompts containing core definitions. All specifications are publicly available in our repository. To control variability in the results, each generated agent was executed three times against the ground-truth test cases, and all reported metrics correspond to the average across runs. A fully automated pipeline with a *pass@k* approach would provide stronger consistency in future studies. **Construct validity.** Metrics derived from LLM-as-a-Judge (clarity, completeness, relevance) may not fully capture nuanced agent behavior. To mitigate non-determinism, each test case was executed three times with results aggregated, and evaluation criteria were explicitly defined per metric to reduce ambiguity in scoring. Additionally, while code quality metrics (Pylint, Flake8) assess stylistic rather than functional correctness, behavioral correctness is separately addressed through the LLM-as-a-Judge evaluation in RQ_4, which executes the generated agents against predefined test cases.

7 Conclusion

This paper presented Yuma-Code, a CLI framework that transforms natural-language specifications into executable LLM agent code through structured requirement elicitation, automatic architecture selection, and multi-agent orchestration. To evaluate Yuma-Code, we introduced AgentGenBench, a benchmark covering four agent architectures (Simple, Tool, Memory, MCP), and compared Yuma-Code against four general-purpose code generation frameworks — Copilot CLI, Gemini CLI, Kiro, and Spec-kit.

Yuma-Code was the only framework to achieve full specification compliance and LLM-as-a-Judge artifact generation across all scenarios, requiring zero manual corrections throughout. Copilot CLI emerged as the closest competitor, producing organized and executable agents, particularly in simpler configurations. Gemini CLI excelled in the Simple scenario but degraded in complex integrations. Kiro and Spec-kit frequently produced fragmented or mocked implementations, limiting their practical utility for agent-specific tasks. The results suggest that treating agent generation as a structured engineering process — encompassing specification, architecture selection, and behavioral evaluation — yields more reliable and verifiable agents than general-purpose code synthesis.

Future Work. AgentGenBench can be extended with additional architectures, task domains, and multi-agent systems. Future studies should include ablation analyses of Yuma-Code's pipeline, a fully automated *pass@k*-based evaluation pipeline, and human validation to complement LLM-as-a-Judge assessments. Yuma-Code can also evolve toward a complete development lifecycle, including deployment and continuous evaluation. Finally, future work will improve the requirements elicitation and prompt generation processes to better balance structural precision, functional correctness, and semantic quality, reducing discrepancies between functional and semantic evaluation metrics.

Artifact Availability

All artifacts and the source code of Yuma-Code are publicly available.[19][20] Experiments were conducted between October and November 2025.

Acknowledgements

This work was supported by Kunumi Institute. The authors thank the institution for its financial support and commitment to advancing scientific research.

[19] https://github.com/Agents4Good/AgentGenBenchCBSoft
[20] https://github.com/Agents4Good/YUMA-code/releases/tag/yuma-code-v1.0.0

References

[1] Leonhard Applis, Yuntong Zhang, Shanchao Liang, Nan Jiang, Lin Tan, and Abhik Roychoudhury. 2025. Unified Software Engineering Agent as AI Software Engineer. arXiv:2506.14683 [cs.SE]

[2] Jacob Austin et al. 2021. Program synthesis with large language models. arXiv:2108.07732 [cs.CL]

[3] Anna Bavaresco et al. 2025. LLMs instead of Human Judges? A Large Scale Empirical Study across 20 NLP Evaluation Tasks. arXiv:2406.18403 [cs.CL]

[4] Lenz Belzner, Thomas Gabor, and Martin Wirsing. 2023. Large language model assisted software engineering: prospects, challenges, and a case study. In *International Conference on Bridging the Gap between AI and Realit*. 355–374. doi:10.1007/978-3-031-46002-9_23

[5] Victor R Basili1 Gianluigi Caldiera and H Dieter Rombach. 1994. The goal question metric approach. *Encyclopedia of software engineering* (1994), 528–532.

[6] Jialun Cao et al. 2026. Rigor, Reliability, and Reproducibility Matter: A Decade-Scale Survey of 572 Code Benchmarks. arXiv:2501.10711 [cs.SE]

[7] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[8] Zhi Chen, Wei Ma, and Lingxiao Jiang. 2026. Beyond Final Code: A Process-Oriented Error Analysis of Software Development Agents in Real-World GitHub Scenarios. arXiv:2503.12374 [cs.SE]

[9] Mike Cohn. 2004. *User Stories Applied: For Agile Software Development*. Addison-Wesley. 6–9 pages.

[10] Xueying Du et al. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the International Conference on Software Engineering*. 982–994. doi:10.1145/3597503.3639219

[11] AutoGen Studio: A No-Code Developer Tool for Building and Debugging Multi-Agent Systems. 2024. Establishing Best Practices for Building Rigorous Agentic Benchmarks. arXiv:2408.15247 [cs.SE]

[12] Dan Hendrycks et al. 2021. Measuring coding challenge competence with APPs. *Arxiv* (2021).

[13] Sirui Hong et al. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. arXiv:2308.00352 [cs.AI]

[14] Juyong Jiang et al. 2024. A survey on large language models for code generation. arXiv:2406.00515

[15] Carlos E. Jimenez et al. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770 [cs.CL]

[16] Naveen Krishnan. 2025. AI Agents: Evolution, Architecture, and Real-World Applications. arXiv:2503.12687 [cs.AI]

[17] Xiao Liu et al. 2025. AgentBench: Evaluating LLMs as Agents. arXiv:2308.03688 [cs.AI]

[18] Yang Liu et al. 2023. G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment. arXiv:2303.16634 [cs.CL]

[19] Yue Liu et al. 2024. Agent Design Pattern Catalogue: A Collection of Architectural Patterns for Foundation Model based Agents. arXiv:2405.10467 [cs.AI]

[20] Ziyang Luo et al. 2025. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv:2306.08568 [cs.CL]

[21] Grégoire Mialon et al. 2023. GAIA: a benchmark for General AI Assistants. arXiv:2311.12983 [cs.CL]

[22] Khouloud Oueslati, Maxime Lamothe, and Foutse Khomh. 2026. RefAgent: A Multi-agent LLM-based Framework for Automatic Software Refactoring. arXiv:2511.03153 [cs.SE]

[23] Jiabin Tang, Tianyu Fan, and Chao Huang. 2025. AutoAgent: A Fully-Automated and Zero-Code Framework for LLM Agents. arXiv:2502.05957 [cs.AI]

[24] Speakeasy Team. 2024. AI Agent Architecture Patterns. https://www.speakeasy.com/mcp/using-mcp/ai-agents/architecture-patterns Accessed: 2025-10-21.

[25] Huanting Wang et al. 2025. AI Agentic Programming: A Survey of Techniques, Challenges, and Opportunities. arXiv:2508.11126 [cs.SE]

[26] Lei Wang et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024).

[27] Zhiheng Xi et al. 2025. The rise and potential of large language model based agents: a survey. *Science China Information Sciences* 68, 2 (2025).

[28] Qinkai Zheng et al. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5673––5684. doi:10.1145/3580305.3599790

[29] Shuyan Zhou et al. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. arXiv:2307.13854 [cs.AI]

[30] Yuxuan Zhu et al. 2025. Establishing Best Practices for Building Rigorous Agentic Benchmarks. arXiv:2507.02825 [cs.AI]