**Empirical Software Engineering manuscript No.**
(will be inserted by the editor)

# An Empirical Study of Testing Practices in Open Source AI Agent Frameworks and Agentic Applications

**Mohammed Mehedi Hasan · Hao Li · Emad Fallahzadeh · Gopi Krishnan Rajbahadur · Bram Adams · Ahmed E. Hassan**

Received: date / Accepted: date

**Abstract** Foundation model (FM)-based AI agents are rapidly gaining adoption across diverse domains, but their inherent non-determinism and non-reproducibility pose testing and quality assurance challenges. While recent benchmarks provide task-level evaluations, there is limited understanding of how developers verify the internal correctness of these agents during development.

To address this gap, we conduct the first large-scale empirical study of testing practices in the AI agent ecosystem, analyzing 39 open-source agent frameworks and 439 agentic applications. We identify ten distinct testing patterns and find that novel, agent-specific methods like *DeepEval* are seldom used (around 1%), while traditional patterns like negative and membership testing are widely adapted to manage FM uncertainty. By mapping these patterns to canonical architectural components of agent frameworks and agentic applications, we uncover a fundamental inversion of testing effort: deterministic components like `Resource Artifacts` (tools) and `Coordination Artifacts` (workflows) consume over 70% of testing effort, while the FM-based `Plan Body` receives less than 5%. Crucially, this reveals a critical blind spot, as the `Trigger` component (prompts) remains neglected, appearing in around 1% of all tests.

Our findings offer the first empirical testing baseline in FM-based agent frameworks and agentic applications, revealing a rational but incomplete adaptation to non-determinism. To address it, framework developers should improve support for novel testing methods, application developers must adopt prompt regression testing, and researchers should explore barriers to adoption. Strengthening these practices is vital for building more robust and dependable AI agents.

**Keywords** AI Agent · LLM Agent · Mining Software Repositories · Agentic Application · Testing

Mohammed Mehedi Hasan and Hao Li and Emad Fallahzadeh and Gopi Krishnan Rajbahadur and Bram Adams and Ahmed E. Hassan
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: {mohammedmehedi.hasan, hao.li,cj79,16gkr1,bram.adams,}@queensu.ca, ahmed@cs.queensu.ca

## 1 Introduction

Autonomous agents have long been a foundational concept in AI, typically defined as systems capable of decision-making and action execution in an environment to fulfill given goals and objectives (Liu et al., 2023a). This classical idea has been supercharged by the advent of foundation models (FMs), giving rise to a new paradigm: FM-based agentic applications. These agentic applications augment an FM "brain" with capabilities such as tool use, planning, and memory, enabling them to tackle complex, high-level goals with minimal human intervention (Hettiarachchi, 2025; Park et al., 2023). Modern agent frameworks further accelerate this trend, enabling developers to build sophisticated single and multi-agent applications that can reason, collaborate, and dynamically adapt their behavior (Wu et al., 2024a).

With the proliferation of agent frameworks and agentic applications, along with the increasingly critical tasks agents are asked to perform, evaluating their effectiveness has become essential. The dominant evaluation culture, inherited from traditional AI, relies on standardized benchmarks that measure task performance (Liu et al., 2023a). For example, AgentBench introduced a multi-dimensional benchmark with eight distinct interactive environments to systematically assess an LLM-as-agent's reasoning and decision-making abilities (Liu et al., 2023a). A similar approach is followed by other popular benchmarks, e.g., GAIA (Mialon et al., 2023), WebArena (Zhou et al., 2023), and MINT (Wang et al., 2023). While valuable for objective comparison, this approach creates a dangerous gap between perceived performance and real-world reliability. Benchmarks primarily test if an agent can succeed on a set of pre-defined tasks, but not whether it is robust, safe, or dependable for an end user's specific needs. For instance, an agent that tops a leaderboard might still perform poorly when faced with edge cases, get stuck in buggy loops, produce hallucinations, or succumb to unhandled faults that benchmarks rarely account for (Weidinger et al., 2025).

To address this reliability gap, practitioners increasingly adopt traditional software testing methods, e.g., unit testing, to verify agent behavior in controlled scenarios. Unlike benchmarks, unit tests not only verify that the software behaves as intended, but also expose edge cases and uncover newly introduced faults in code chunks that previously worked properly (Niedermayr et al., 2016). However, adapting traditional testing practices to FM-based agent frameworks and agentic applications is tricky due to the inherent non-determinism and non-reproducibility of the underlying models (Hassan et al., 2024).

Surprisingly, this critical area remains almost entirely unexplored. While research has focused extensively on agent architectures, capabilities, and benchmarks, there is a clear lack of empirical evidence on how practitioners actually test agent frameworks and agentic applications. We found no comprehensive studies examining the state-of-the-art testing practices for open-source agents: how developers structure tests, verify non-deterministic outputs, and which parts of the agent they prioritize while testing. These gaps are concerning, given that the potential impact of untested or unverified agents can be poor in real-world use cases. For example, previous studies have already reported that the performance of FM-based systems can deteriorate during model upgrades, and without testing the prompts and managing the versions, it is difficult to identify such silent performance degradation (Ma et al., 2024). Similarly, prior studies have reported at least seven error patterns while invoking tools in FM-based systems (Kokane et al., 2025), which can also lead to production issues if

not appropriately tested during the development phase. To complement the findings from researchers, we have also observed that different components (e.g., prompts, tools) behave differently in different FMs or even on different versions of the same FM.[1]

The challenge is further amplified by the rapid evolution of these agent frameworks and agentic applications. Practitioners are confronted with a constant flux of new, implementation-specific components, e.g., from novel tool invocation protocols and multi-agent communication standards (MCP, A2A, ACP) to evolving memory and planning modules (Hasan et al., 2025; Ehtesham et al., 2025). This architectural evolution makes it exceedingly difficult to establish durable testing strategies; what is tested today may become obsolete or change fundamentally tomorrow. As established in software engineering research, the key to managing such complexity is to map these volatile, concrete components to a stable, canonical or conceptual architectural model, which provides a durable foundation for reasoning about system quality and test coverage (Lukyanenko et al., 2024). While prior work has proposed valuable taxonomies that categorize emerging agent components (Händler, 2023a), we have not seen any work grounding the architecture component to any conceptual architecture till date.

To fill these critical gaps, we conduct the first large-scale empirical study of unit testing practices in the agent ecosystem. We analyze 39 agent frameworks and 439 agentic applications, examining how practitioners arrange tests and assert correctness. Then we map the state-of-the-art agent architecture component to an existing conceptual architecture framework Boissier et al. (2020) and identify component-specific testing patterns. Our study answers the following research questions (RQs):

**RQ1: What testing practices are commonly adopted by practitioners to evaluate open-source AI agent frameworks and agentic applications?**

**Motivation**: The growing adoption of FM-based agents in business-critical domains has intensified the need for robust quality assurance. However, the inherent non-determinism and non-reproducibility of foundation models (FMs) significantly complicate the application of traditional, deterministic testing techniques (Hassan et al., 2024). These challenges require practitioners to adopt alternative testing strategies that can accommodate unpredictable behavior during both test arrangement and result verification.

Despite extensive research into testing methodologies within traditional software engineering (Zhu et al., 2025; Zamprogno et al., 2022) and machine learning applications (Openja et al., 2024), the FM-based agent ecosystem remains underexplored. This study fills that gap by conducting the first large-scale empirical investigation of unit testing practices in open-source agent frameworks and applications. By systematically analyzing how practitioners structure and verify test functions under these constraints, the work provides a foundational catalog of testing strategies that can guide the development of more reliable and resilient agentic systems.

**Findings**: We find a crucial split in practitioner strategy. On the one hand, novel patterns designed explicitly for this paradigm, such as *DeepEval* and *Hyperparameter Control*, see very low adoption, suggesting a significant knowledge or awareness gap. On the other hand, to compensate for this, practitioners strategically adapt familiar,

[1] `https://community.openai.com/t/gpt-4o-vs-gpt-4-turbo-function-calling/750493` and `https://www.reddit.com/r/ChatGPT/comments/1idghel/was_anyone_elses_experience_with_gpt4o_completely`

battle-tested, less strict verification patterns, e.g., Membership Testing, Mock Assertion, and Negative Testing, to gracefully handle non-determinism and avoid the test fragility caused by strict assertions.

**RQ2: How do these testing practices map to the architectural components of the agent frameworks and agentic applications?**

**Motivation**: Agent ecosystem is a complex ecosystem of interacting components, from memory and tools to planning and coordination artifacts (Hettiarachchi, 2025). Each of these components introduces distinct behaviors and potential failure modes, demanding tailored testing strategies. Understanding how testing effort is distributed across this heterogeneous architecture is vital for assessing whether critical components receive sufficient attention and for supporting the development of more systematic and effective testing practices. A comprehensive empirical understanding of component-wise testing patterns is also essential for helping practitioners formulate robust testing strategies for new agent systems.

A significant challenge for empirical study, however, is the ecosystem's rapid evolution, with standards for core functions like tool use (MCP) Hasan et al. (2025), communication (A2A, ACP, ANP) (Ehtesham et al., 2025), and memory emerging and changing continuously. To create a durable analysis that transcends these rapid implementations, we opt to map observed components to a stable, canonical architectural model, e.g., JaCaMo framework Boissier et al. (2020), which will allow us to assess whether testing effort is distributed effectively across fundamental roles, supporting the development of more systematic testing practices. To this end, our second research question first maps the observed components in the test functions to canonical components of conceptual agent architecture and then investigates the relationship between the testing patterns identified in RQ1 and the underlying components of agent frameworks and applications.

**Findings**: We identify 13 canonical components that receive unit-testing attention in agent frameworks and agentic applications. First, we observe a strategic inversion of testing efforts in agent frameworks and agentic applications compared to traditional ML applications: instead of focusing on the model itself, developers concentrate on deterministic infrastructure such as `Resource Artifacts` (tools, parsers), which account for 29.7% of tests in frameworks and 40.1% in applications. Second, we identify a critical testing blind spot: the `Trigger` component (prompts) is dangerously under-tested, appearing in around 1% of test functions, which can introduce significant risks of silent failures and performance degradation as the underlying foundation models evolve.

The main contributions of our paper are as follows:

- **A Curated Dataset**: The first large-scale, curated dataset of 39 agent frameworks, 439 agentic applications, and their corresponding test functions, providing a foundational resource for future research.
- **A Taxonomy of Agent Testing Patterns**: A comprehensive taxonomy of 10 testing patterns, including novel and adapted traditional techniques, used to manage the challenges of testing agentic systems.
- **First Comparative Baseline**: The first empirical baseline of testing practices in the agent ecosystem in comparison with traditional software engineering and ML applications' testing patterns.
- **Canonical Architectural Component List**: An extensible catalog of 13 canonical components built on the classical JaCaMo framework (Boissier et al., 2020)

and recent FM-based agent taxonomies. Twelve components correspond directly to JaCaMo, while one additional component (i.e., `Registry`) is introduced based on contemporary practice. This catalog provides a reusable reference model for analyzing and testing future agent frameworks and agentic applications.
- **Component-wise Testing Patterns**: A comprehensive mapping of testing patterns to canonical components of agent frameworks and agentic applications, which can be leveraged by practitioners to select appropriate testing strategies for existing components and to design tests for new components.

## 2 A Motivational Example

John, a senior developer with a background in traditional software engineering, recently joined an ambitious AI startup. His first assignment is to lead the development of an agentic application: a playful, interactive voice assistant for children. The agent's core function is to entertain by dynamically generating jokes and bedtime stories using a state-of-the-art Foundation Model (FM).

**Scenario 1: It Just Works**. Using a popular open-source agent framework, John quickly builds a prototype. The framework provides seamless integration with the FM. When a child asks for a story, the agent prompts the FM, which returns a creative and engaging narrative. The initial results are impressive, and the team moves forward with deployment, confident in the power of their new AI-driven architecture.

**Scenario 2: The Unseen Decay**. Weeks after launch, user feedback reveals a degradation in quality. Jokes are becoming boring, and stories occasionally lack coherence. John investigates and finds the root cause: a silent, unannounced update to the underlying FM has altered its interpretation of their prompts. The application is performing, yet the user-facing quality has declined. He realizes he has no automated way to detect this semantic drift, e.g., no tests. He wonders: How do other teams test for this?

**Scenario 3: The Component Stack-Up**. Things become more complicated with the product team introducing two new features:

- **Contextual Follow-up**: Children should be able to ask for a story "just like the one from last night." This feature requires integrating a VectorDB for Retrieval-Augmented Generation (RAG) to maintain conversational history and ensure story similarity (Lewis et al., 2020).
- **Real-World Q&A**: During a story about Cinderella, a child might ask, "How much is a ticket to Disneyland?" This requires the agent to use a third-party search tool, e.g., an MCP server (Hasan et al., 2025).

John's testing challenges have now multiplied. Not only does he need to validate the non-deterministic output of the FM, but he must now also evaluate distinct components:

- **The RAG system**: How can he write a test to verify that a newly generated story is "similar and relevant" to a previous one? What metrics define "similarity"?
- **The MCP Server**: How does he test a third-party MCP Server for accuracy, fairness, and truthfulness? The server is a black box that could change its behavior at any time.

**Scenario 4: The Search for a Standard**. Coming from traditional software engineering, John is familiar with mocking, stubbing, and integration testing. He attempts to apply these concepts, but they feel ill-suited for a system with many non-deterministic and independently evolving parts. He investigates how other, more mature open-source agentic applications and frameworks handle this. He dives into the GitHub repositories of popular projects, looking for their testing suites, hoping to find established best practices or community-standard tools for evaluating these complex components.

**Scenario 5: The Challenge**. John's search leaves him perplexed. He finds a scattered landscape of ad-hoc scripts, manual evaluation checklists, and a heavy reliance on costly, slow, end-to-end testing that offers little insight into which component failed. There appears to be no standardized methodology for ensuring the quality and reliability of agentic applications. This leads him to a critical set of questions that must be answered to mature the field of agent development:

1. What testing practices are commonly adopted by practitioners to evaluate open-source AI agent frameworks and agentic applications? **(explored in RQ1)**
2. How are the components of AI agent frameworks and agentic applications evaluated?**(explored in RQ2)**

## 3 Background

This section establishes the foundational concepts underpinning our study. We first define the core principles of software testing, then describe the emerging agent ecosystem, and finally, present a canonical agent architecture that provides a stable model for analyzing testing practices in this rapidly evolving domain.

### 3.1 Foundations of Software Testing

#### *3.1.1 Testing*

In software engineering, software testing is a core part of quality assurance. It is empirically defined as the systematic execution of a program or system with the explicit objective of identifying and isolating defects (Myers, 2006). The fundamental purpose of this process is to methodically evaluate the functional and non-functional attributes of the software, thereby ensuring its adherence to specified requirements prior to its release and implementation.

#### *3.1.2 Subject Under Test (SUT)*

The Subject Under Test (SUT) is the specific software artifact, e.g., a function, class, module, or entire service, that is being validated in the testing process (Meszaros, 2007). In our study of the agent ecosystem, SUTs range from low-level resource handlers and parsers to complex, high-level agentic workflows.

#### *3.1.3 Test Function*

A test function, also known as a test case, is the atomic unit of verification. It is an executable procedure designed to validate a specific behavior or functionality of the SUT (Van Rompaey and Demeyer, 2008). In practice, a test function initializes a specific state or fixture, executes a command on the SUT, and verifies the outcome using one or more assertions (Tao, 2009).

#### *3.1.4 Arrange-Act-Assert (AAA)*

The Arrange-Act-Assert (AAA) pattern is a widely adopted convention for structuring test functions to enhance their readability and maintainability (Wei et al., 2022a). The pattern divides a test into three distinct phases: arranging the preconditions and inputs, acting on the SUT, and asserting that the outcome meets expectations. We adopt AAA as the conceptual framework for analyzing and classifying the testing practices in our study.

#### *3.1.5 Testing Patterns*

Testing patterns are recurring, expert solutions to common problems in test design and implementation (Meszaros, 2007). These patterns provide proven strategies for tasks such as isolating the SUT from its dependencies, managing test data, and verifying complex outcomes. For instance, the use of test doubles (e.g., mocks or stubs) is a classic pattern for isolating a SUT (Zhu et al., 2025). These patterns improve the manageability and reliability of a test suite.

### 3.2 The Agent Ecosystem

#### *3.2.1 Agent*

An agent is traditionally defined as an autonomous entity that executes decisions and actions within a given environment to realize predetermined objectives (Liu et al., 2023a). This conceptualization has its roots in established paradigms such as the Belief-Desire-Intention (BDI) framework (Rao et al., 1995). The capabilities of such agents have been significantly amplified by the development of Foundation Models (FMs). In contemporary implementations, agents leverage a core FM that functions as a cognitive engine, which is supplemented by essential modules for memory, strategic planning, and the utilization of external tools. This integration empowers the agents to address intricate and unstructured challenges with a high degree of autonomy (Hettiarachchi, 2025; Park et al., 2023).

#### *3.2.2 Agent Framework*

Agent frameworks are software libraries that provide a structured environment for building, orchestrating, and deploying agentic systems. Frameworks like AutoGen (Wu et al., 2024a) and CAMEL (Li et al., 2023) abstract away the underlying complexities of state management, tool integration, and multi-agent communication. By offering high-level APIs for defining agent roles and workflows, they accelerate the development of sophisticated single and multi-agent applications (Händler, 2023b).

#### *3.2.3 Agentic Application*

In this study, an agentic application is a software system built on top of agent frameworks. These applications leverage FMs to perform complex tasks that often require interaction with external systems, such as a storyteller agent that autonomously invokes APIs for searching characters, booking tickets, and querying vectorDBs as shown in the motivational example of Section 2. Such applications use the capabilities provided by the agent frameworks described above to manage their operational logic.

### 3.3 Canonical Agent Architecture

To create a stable and durable analytical foundation that transcends implementation-specific details, we map the components observed in our dataset to the canonical, multi-agent systems architecture defined by the JaCaMo framework (Boissier et al., 2020). This model organizes components into three dimensions: the Agent, the Environment, and the Organization. We supplement this classical model with newer concepts like registries, which have become central to modern agent architectures (Liu et al., 2025).

#### *3.3.1 Belief Base*

The Belief Base contains the set of information, or beliefs, that an agent holds about itself and its environment at a given time. These beliefs, which may not always be complete or correct, form the agent's knowledge foundation for reasoning and decision-making (Boissier et al., 2020). In FM-based agents, this corresponds to memory systems, vector stores, and other persistent data sources.

#### *3.3.2 Goal*

A Goal represents a desired state of the world that an agent aims to achieve. Goals are the primary drivers of an agent's behavior, guiding its deliberation and planning processes to move from its current state to a desired one (Masterman et al., 2024).

#### *3.3.3 Plan*

A Plan is a course of action that an agent can execute to achieve a goal. A plan is typically composed of a trigger, a context, and a body.

#### *3.3.4 Trigger*

The triggering event that activates a plan. In FM-based agents, the initial user request or prompt is the primary trigger for invoking a plan (Boissier et al., 2020).

#### *3.3.5 Context*

Supplemental, just-in-time information that helps an agent select the most appropriate plan from a set of available options. This is analogous to the documents and data supplied in Retrieval-Augmented Generation (RAG) to ground an FM's response (Boissier et al., 2020).

#### *3.3.6 Plan Body*

The sequence of actions or reasoning steps that constitute the plan. In modern agents, the plan body is often dynamically generated by an FM, which uses its reasoning capabilities to break down a high-level goal into concrete, executable steps (Cheng et al., 2024; Shen et al., 2024).

#### *3.3.7 Internal Action*

A computation performed within the agent's own boundary, such as updating a counter or transforming data, with no external effect (Boissier et al., 2020).

#### *3.3.8 External Action*

An operation that affects the agent's external environment, such as making an API call, executing code, or controlling a physical device (Boissier et al., 2020).

#### *3.3.9 Communicative Action*

A message exchanged between agents or between an agent and a user. These actions are fundamental to coordination and collaboration in multi-agent systems (Boissier et al., 2020).

#### *3.3.10 Resource Artifact*

Resource Artifacts are passive, reusable entities in the environment that agents can utilize to perform their tasks. Examples include shared resources like a database connection, a file, or a wrapper for an external API or tool (Boissier et al., 2020).

#### *3.3.11 Coordination Artifact*

Coordination Artifacts are entities specifically designed to manage and synchronize the interactions among multiple agents. Examples include shared data structures like message queues, blackboards, or event buses that facilitate orderly collaboration (Boissier et al., 2020).

#### *3.3.12 Boundary Artifact*

Boundary Artifacts are components that serve as a bridge between the agent system and the outside world. They enable agents to interact with external entities that are not part of the multi-agent system, such as a human user via a graphical user interface (GUI) or another software system (Boissier et al., 2020).

#### *3.3.13 Observable Property & Signal*

These are mechanisms through which an agent perceives the state of an artifact. An Observable Property is a piece of data representing an artifact's state, which an agent can read. A Signal is an event emitted by an artifact to notify observing agents of a significant change or occurrence (Boissier et al., 2020). In modern systems, these correspond to logs, metrics, and return values.

#### *3.3.14 Registry*

While not part of the original JaCaMo model, the Registry has emerged as a critical component in modern agent ecosystems (Liu et al., 2025). A Registry is a centralized catalog where developers can publish, discover, and reuse agents or tools. For instance, MCP registries allow for the standardized discovery and invocation of tools across different frameworks, promoting interoperability and reuse (Hasan et al., 2025).

## 4 Related Work

### 4.1 Testing Ecosystem

#### *4.1.1 Testing in OSS Ecosystem*

Software testing is a foundational software engineering practice aimed at verifying quality and finding defects. A classic definition by Myers emphasizes the mindset: "Testing is the process of executing a program with the intent of finding errors" (Myers, 2006). Researchers have extensively studied how developers write and maintain tests in traditional software projects (e.g., Java, Python, JavaScript ecosystems), yielding common findings on test design, structure, and effectiveness. Meszaros's xUnit Test Patterns catalog introduced common patterns in test code (Meszaros, 2007).

A practical lens for examining these techniques is the AAA (Arrange, Act, Assert) paradigm, which structures tests into clear phases of setup, execution, and verification (Wei et al., 2022a). While empirical studies show inconsistent adoption of specific design-level patterns (e.g., only 24% of projects applied certain maintainability-focused patterns (Gonzalez et al., 2017)), the structural AAA pattern has become ubiquitous, with 77% of Java unit tests adhering to its format to enhance readability (Wei et al., 2025). This review adopts the AAA structure to survey established testing techniques and foreshadow their application and limitations in the context of AI agents.

For the `Arrange` phase, we have seen different testing patterns in the OSS domain in the literature. For example, the use of test doubles (e.g., Mocks, Stubs, Fakes) eliminates the need for complex setup procedures, e.g., practitioners no longer need to set up databases for testing or simulate network failures through physical interventions like unplugging network cables (Zhu et al., 2023). Similarly, parameterized testing enables practitioners to execute the same test logic with multiple inputs and has been proven to run test functions 30 times faster than the traditional approach (Kampmann and Zeller, 2019).

In the `Assert` phase, OSS testing relies on verifiable outcomes. Studies have cataloged 12 assertion patterns, with equality checks being common (39.3% in JavaScript projects) (Zamprogno et al., 2022), mock assertion techniques, where practitioners often assert that certain calls were made on the mock (interaction-based verification), rather than asserting a returned value has also gained popularity with the test doubles is found in 41% of the test functions and 50% of the defects detected by mock assertion are missed by other assertion techniques (Zhu et al., 2025). The importance of assertion is also underscored by recent work on automatically generating asserts using ML and FMs (Fontes and Gay, 2023; Wang et al., 2024a). However, until now, it is not known how these structural patterns used for `Arrange` and verification patterns used for `Assert` are helping practitioners to test the agent frameworks and agentic applications.

#### *4.1.2 Testing in ML Ecosystem*

Moving beyond traditional code-centric testing, the Machine Learning (ML) ecosystem introduces the challenge of validating systems whose behavior is learned from data rather than explicitly programmed. This marks a fundamental shift in focus from verifying static code logic to testing dynamic model behavior, data pipelines, and performance over time (Zhang et al., 2020). In response, researchers have proposed various testing frameworks tailored to ML contexts, including tools like DeepXplore (Pei et al., 2017), TEST4Deep (Nishi et al., 2018), and FairTest (Tramer et al., 2017). These frameworks support the validation of models under diverse inputs and conditions, addressing concerns such as fairness, robustness, and interpretability.

Apart from this, a new suite of validation patterns and frameworks emerged. Techniques like *Oracle Approximation* compare a model's output against a simpler, trusted heuristic (Nejadgholi and Yang, 2019), while *Value Range Analysis* ensures predictions fall within plausible bounds (Sahoo et al., 2021). Recent empirical work confirms this evolution, identifying nine high-level testing patterns used by practitioners to verify 16 distinct ML-specific properties, from model performance to data quality (Openja et al., 2024). However, current studies have not explored how the tests are being organized in the ML ecosystem, e.g., from `Arrange` point of view. Also, while prior work in ML testing has identified verification patterns like Value Range Analysis, it remains unknown how these patterns are being adapted to test the unique, composite architectures of FM-based agent frameworks and agentic applications, which blend traditional code with non-deterministic components.

### 4.2 Agent Ecosystem

#### *4.2.1 Classical to FM-Based Agents*

Building on the foundational BDI agent paradigm, which conceptualized agents as autonomous entities equipped with Beliefs, Desires, Intentions, and a plan library to support real-time decision-making, researchers expanded these concepts through formal models like OBDI that ground these mental constructs in computational semantics (Rao et al., 1995). More recent JaCaMo framework implements the core components of a classical BDI agent demonstrating a plan library (a set of pre-defined procedures for achieving goals), a belief base (a data structure to store beliefs), and

an interpreter that runs a continuous reasoning cycle: perceive events, update beliefs, deliberate on desires to form intentions, and execute a relevant plan (Boissier et al., 2020). However, the advent of Foundation Models (FMs) has introduced a paradigm shift, replacing explicitly programmed logic of plan library with an FM as the core reasoning engine (Guo et al., 2024) The architecture of these FM-based agents typically integrates core components like an FM-based reasoning engine, clearly defined objectives, memory systems for historical context, an action module to interact with external tools and environments, and a reflective "Rethink Module" for continuous improvement enabling agents to utilize diverse tools such as APIs, robotic control systems, and code interpreters, significantly broadening their applicability across various domains (Masterman et al., 2024; Cheng et al., 2024; Wang et al., 2024b)

#### *4.2.2 Agent Frameworks*

To manage the complexity of building sophisticated FM-based agents, a new ecosystem of agent frameworks has emerged. For example, CAMEL framework leverages a role-playing paradigm with inception prompting, a prompting technique that enables agents to prompt each other to solve tasks, to generate large-scale conversational datasets for studying agent cooperation and "cognitive" behaviors (Li et al., 2023). On the other hand, AutoGen introduces conversable agents and conversation programming to abstract complex workflows into agent-to-agent dialogues, supporting mixed LLM, human, and tool interactions across diverse domains (Wu et al., 2023). MetaGPT encodes Standardized Operating Procedures (SOPs) into prompt sequences and uses an assembly-line model to decompose tasks among specialized agents, improving consistency and reducing hallucination cascades (Hong et al., 2023). Despite their strengths, their published evaluations focus on end-to-end application performance and benchmarking, leaving how they internally test core components (e.g., memory management, planning, tool invocation, and self-reflection) largely unexplored.

#### *4.2.3 Evaluating Agent Frameworks and Agentic Systems*

Current approaches of evaluating AI agents predominantly focus on task-oriented performance, relying on standardized benchmarks like AgentBench, GAIA, and WebArena to measure an agent's problem-solving capabilities (Liu et al., 2023a; Mialon et al., 2023; Zhou et al., 2023). These benchmarks offer structured tasks for assessing agent capabilities such as planning, tool use, and decision-making. However, recent literature has increasingly emphasized the limitations of benchmark-driven evaluation, particularly in addressing the practical challenges posed by foundation model (FM)-based agents (Hassan et al., 2024). Specifically, issues such as non-determinism, non-reproducibility, and prompt sensitivity have drawn attention to the need for more robust quality assurance practices (Hassan et al., 2024; Ma et al., 2024). Moreover, recent efforts have highlighted vulnerabilities such as tool failures (Kokane et al., 2025), further underscoring the limitations of relying solely on high-level benchmarks for ensuring system reliability.

Despite these growing concerns, to the best of our knowledge, there is no comprehensive research on the use of quality assurance activities, e.g., unit testing, in agent frameworks and agentic applications till date. This study aims to bridge that gap

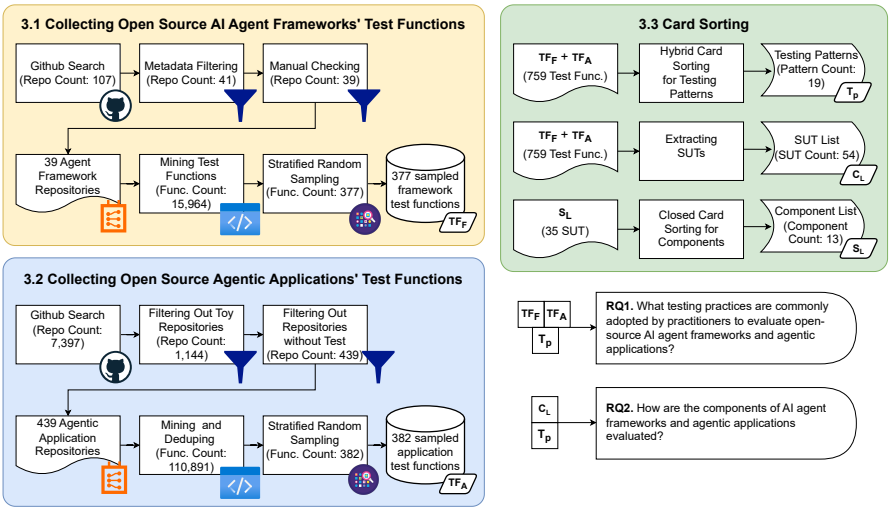



Fig. 1: Overview of the Research Method

by conducting the first large-scale empirical investigation of unit testing practices in open-source agent frameworks and the agentic applications built upon them.

## 5 Methodology

In this section, we describe the methodology used to investigate testing practices in AI agent frameworks and agentic applications. In this study, we adopt a multi-stage empirical approach consisting of three structured phases: (i) collecting test functions from open-source AI agent frameworks, (ii) collecting test functions from open-source agentic applications, and (iii) conducting a systematic qualitative analysis of the test functions through manual card sorting to identify different testing practices and components under test. An overview of the methodological workflow is presented in Figure 1. In the following, we provide a detailed explanation of each step.

### 5.1 Collecting Open-Source AI Agent Frameworks' Test Functions

To analyze the testing practices of AI agent frameworks, we first identify representative agent framework repositories and systematically extract their test functions. To accomplish these objectives, we employ multiple sub-steps, which are discussed below in chronological order.

#### *5.1.1 GitHub Search*

Following prior empirical studies on open-source software curation in emerging ML/AI domains (Gonzalez et al., 2020; Li and Bezemer, 2025), we leverage GitHub's keyword-based repository search API using domain-specific keywords:

Table 1: AI Agent Framework: Repository Metadata Filtering

| Filter Condition | Repository Count |
|---|---|
| Mined from GitHub | 107 |
| Contributor count $\geq 2$ | 104 |
| Star count $\geq 1000$ | 67 |
| Language: Python | 46 |
| Number of test files $\geq 1$ | 41 |
| Manual Checking | 39 |

"`AI AND agent AND framework`", "`LLM-based AND agent AND framework`", "`LLM AND agent AND library`", "`multi-agent orchestration framework`", "`LLM powered agents AND framework`"

We conducted the search on Jun 14, 2025, and retrieved a total of 107 agent framework repositories from this search step.

*5.1.2 Metadata Filtering*

GitHub's repository search API retrieves repositories based on matches with the repository name, description, or README file which can often surface non-representative or experimental repositories (Kalliamvakou et al., 2014; Munaiah et al., 2017). To address this, we filter repositories using the GitHub Repository and Contributor APIs.[2] The filtering steps, shown in Table 1, are as follows:

*Contributor and Popularity Filtering.* To focus on well-maintained frameworks, we first filter repositories based on contributor count and popularity metrics. Prior research has highlighted that the number of contributors is a strong indicator of repository sustainability and adoption (Han et al., 2019). Therefore, we only retain repositories with at least two contributors.

Additionally, GitHub stars are commonly used as a proxy for repository popularity in empirical studies (Openja et al., 2024; Gonzalez et al., 2020; Munaiah et al., 2017) and we include only repositories with at least 1,000 stars. This step reduces the dataset to 67 repositories.

*Language-Specific Filtering.* Since the majority of AI agent frameworks are implemented in Python, we restrict our selection to Python-based repositories. This decision aligns with previous empirical studies in the ML domain, which focused on Python-based projects due to their prevalence in AI research (Openja et al., 2024). Consequently, 21 non-Python repositories are removed, leaving 46 repositories.

*Ensuring Presence of Test Files.* As the primary objective of this study is to analyze testing practices, it is essential to ensure that all selected repositories contain at least one test file. In our study, we consider a file to be a test file if its filename started with the **test**_ prefix, following conventions from previous studies (Openja et al., 2024). We apply this filename-based test file detection technique and remove 5 repositories that do not contain any test files. This step results in 41 repositories.

[2] `https://docs.github.com/en/rest`

#### *5.1.3 Manual Checking*

Although metadata-based filtering helps to remove many irrelevant repositories, we cannot completely rely on this step to identify agent frameworks. For instance, some repositories may include agent-related keywords without implementing an AI agent framework. Therefore, we conduct a manual verification step to ensure that only genuine agent frameworks are retained.

The first and second authors evaluate each repository independently based on the following criteria:

- The repository publishes a PyPI package and includes documentation or examples demonstrating how to build agents using the framework.
- If no PyPI package is available, the README file must explicitly describe agent-related capabilities and provide usage instructions.
- In the absence of detailed documentation in the README, the repository must reference an official website with verifiable information about the framework's agent-supporting features.

Two repositories fail to meet those criteria, resulting in a final curated set of 39 AI agent frameworks. Table 2 consolidates the names and statistics of these frameworks.

#### *5.1.4 Mining Test Functions*

To systematically analyze the testing patterns, we need to extract the test functions from the 39 curated AI agent frameworks. Given the volume and complexity of these repositories, manual extraction was infeasible. We therefore develop an automated parser that follows standard Python testing conventions (Okken, 2022) and prior research (Bodea, 2022).

We define a test function as a Python function that meets both of the following criteria:

- The function is located in a file whose name matches the pattern `test_*.py`.
- The function name begins with the prefix `test_`, conforming to pytest and unittest discovery mechanisms.[3]

We leverage **Abstract Syntax Tree** (AST), a structured representation of source code that preserves its syntactic hierarchy (Cui et al., 2010), to extract test functions by traversing the source code of each agent framework programmatically (Spirin et al., 2021). We extract a total of 15,964 test functions from 1,981 test files. While these test functions are unstructured data, we need a database that can facilitate storing and analyzing unstructured data efficiently. We use Elasticsearch,[4] an open-source search engine database, to store and manage these test functions efficiently for further analysis and searching.

#### *5.1.5 Stratified Random Sampling*

Given the scale of our dataset, i.e., 15,964 test functions across 39 agent frameworks, manually analyzing every function is infeasible. Additionally, applying simple random sampling can result in over-representation from larger repositories and

[3] `https://docs.pytest.org/en/stable/how-to/unittest.html`

[4] `https://github.com/elastic/elasticsearch`

Table 2: Overview of AI agent frameworks and their GitHub statistics

| AI agent framework | # Stars (k) | # Contributors | # Test files | # Test functions |
|---|---|---|---|---|
| FoundationAgents/MetaGPT | 56.4 | 146 | 240 | 964 |
| microsoft/autogen | 45.9 | 551 | 80 | 1041 |
| crewAIInc/crewAI | 32.9 | 248 | 50 | 588 |
| agno-agi/agno | 28.2 | 227 | 158 | 1697 |
| huggingface/smolagents | 20.1 | 157 | 18 | 463 |
| openai/swarm | 19.9 | 13 | 2 | 11 |
| letta-ai/letta | 16.8 | 135 | 30 | 650 |
| eosphoros-ai/DB-GPT | 16.8 | 146 | 73 | 728 |
| TransformerOptimus/SuperAGI | 16.4 | 73 | 114 | 466 |
| raga-ai-hub/RagaAI-Catalyst | 16.2 | 26 | 17 | 87 |
| langchain-ai/langgraph | 14.3 | 229 | 43 | 906 |
| camel-ai/camel | 12.9 | 148 | 221 | 1507 |
| openai/openai-agents-python | 11.4 | 96 | 61 | 563 |
| pydantic/pydantic-ai | 10.2 | 161 | 74 | 1096 |
| QwenLM/Qwen-Agent | 9.7 | 28 | 22 | 48 |
| Upsonic/Upsonic | 7.5 | 24 | 5 | 24 |
| crestalnetwork/intentkit | 6.4 | 15 | 2 | 17 |
| livekit/agents | 6.3 | 150 | 15 | 137 |
| lavague-ai/LaVague | 6.1 | 25 | 2 | 1 |
| awslabs/agent-squad | 6.0 | 23 | 22 | 312 |
| superduper-io/superduper | 5.1 | 48 | 65 | 318 |
| kyegomez/swarms | 4.9 | 44 | 54 | 645 |
| MervinPraison/PraisonAI | 4.8 | 20 | 33 | 241 |
| AgentOps-AI/agentops | 4.5 | 42 | 28 | 337 |
| VRSEN/agency-swarm | 3.7 | 19 | 5 | 78 |
| SylphAI-Inc/AdalFlow | 3.3 | 26 | 37 | 299 |
| cheshire-cat-ai/core | 2.8 | 86 | 45 | 197 |
| Intelligent-Internet/ii-agent | 2.4 | 6 | 5 | 67 |
| griptape-ai/griptape | 2.3 | 37 | 256 | 1242 |
| dot-agent/nextpy | 2.3 | 12 | 88 | 828 |
| InternLM/lagent | 2.1 | 33 | 5 | 8 |
| trypromptly/LLMStack | 2.0 | 8 | 15 | 57 |
| melih-unsal/DemoGPT | 1.8 | 4 | 2 | 5 |
| openlit/openlit | 1.6 | 36 | 20 | 43 |
| agentuniverse-ai/agentUniverse | 1.5 | 27 | 30 | 70 |
| fetchai/uAgents | 1.4 | 60 | 15 | 136 |
| patched-codes/patchwork | 1.4 | 15 | 18 | 65 |
| Div99/agent-protocol | 1.2 | 15 | 11 | 22 |
| **Total** | – | – | **1,981** | **15,964** |

under-representation from smaller ones. To ensure balanced coverage, we employ *stratified random sampling*, where each repository is treated as a distinct stratum and contributes a proportionate number of samples (Zhao et al., 2024).

To determine an appropriate sample size, we followed statistical practices from prior empirical software engineering studies (Openja et al., 2024). Using a 95% confidence level and a 5% margin of error, we calculate the required sample size using standard formulas for estimating proportions in finite populations, yielding a target of 377 test functions.

We then apply stratified random sampling, whereby these 377 test functions are randomly selected within each agent framework repository in proportion to the total number of test functions that repository contains. This ensures that the final sample

is both representative of the overall dataset and preserves the distribution of test functions across different agent frameworks.

## 5.2 Collecting Open-Source Agentic Applications' Test Functions

After collecting test functions from open-source AI agent frameworks, as described in Section 5.1, the next step is to mine test functions from open-source agentic applications, i.e., standalone projects built using one or more of the AI agent frameworks. To identify such agentic applications, we perform dependency analysis and repository mining using the curated framework list shown in Table 2 as a seed. Specifically, we look for open-source repositories that import these frameworks through explicit mentions in their source code.

In the remainder of this section, we describe our process for mining test functions from agentic applications.

### *5.2.1 GitHub Search*

To discover open-source agentic applications, we perform GitHub Code Search using framework-specific import patterns. From the 39 agent frameworks listed in Table 2, we observe that six do not expose software development kits (SDKs) or importable libraries. These frameworks primarily support GUI-based agent construction (e.g., drag-and-drop interfaces) and do not facilitate direct code-level integration. As a result, applications built on top of these frameworks could not be reliably identified via code analysis and are therefore omitted from this phase.

For each of the remaining 33 frameworks, we manually derive the Python import statements by inspecting their documentation, source code, and published examples. These statements are then used as search queries via the GitHub Code Search API,[5] which returns matches at the file level along with associated repository metadata. Table 3 summarizes the frameworks used, the corresponding import statements employed in our GitHub code search, and the initially identified repository count and the eventual number of agentic application repositories downloaded after applying different filters (see Section 5.2.2).

### *5.2.2 Filtering Toy Repositories*

We collect a list of 7,397 repositories importing at least one agent framework of Table 3 from an initial code search in GitHub. Similar to Section 5.1.2, we observe toy projects and personal one-time proof-of-concept (PoC) repositories in this list. Since the number of repositories is significantly higher than that of framework repositories, we apply additional metadata-based criteria, e.g., fork status, repository lifetime (Hassan and Rahman, 2022), and commit frequency (Munaiah et al., 2017), to filter out toy repositories in this set. We use the following threshold for each of the criteria:

- To avoid duplicate test functions, we exclude all repository forks or clones of another repository, following previous empirical studies about testing (Hassan and Rahman, 2022).

[5] `https://docs.github.com/en/rest/search/search`

Table 3: Framework-wise import patterns, GitHub Code Search results, number of unique repositories in the search result, and downloaded repository counts after filtering toy repositories sorted by #Search result.

| **Framework name** | **# Import statement(s)** | **# Search results** | **# Repos** | **# Filtered repos** |
|---|---|---|---|---|
| langchain-ai/langgraph | `from langgraph.` | 6,720 | 904 | 67 |
| openai/openai-agents-python | `from agents import` | 5,208 | 771 | 41 |
| camel-ai/camel | `from camel.` | 3,488 | 154 | 17 |
| crewAIInc/crewAI | `from crewai` | 2,964 | 832 | 45 |
| huggingface/smolagents | `from smolagents` | 2,672 | 536 | 39 |
| microsoft/autogen | `from autogen_agentchat.`<br>`from autogen_ext.` | 2,572 | 426 | 40 |
| agno-agi/agno | `from agno.` | 2,228 | 343 | 15 |
| pydantic/pydantic-ai | `from pydantic_ai import` | 1,428 | 652 | 51 |
| cheshire-cat-ai/core | `from cat.` | 1,656 | 342 | 19 |
| FoundationAgents/MetaGPT | `from metagpt.` | 1,024 | 122 | 14 |
| fetchai/uAgents | `from uagents import` | 912 | 340 | 9 |
| QwenLM/Qwen-Agent | `from qwen_agent.` | 808 | 108 | 6 |
| griptape-ai/griptape | `from griptape.` | 820 | 78 | 13 |
| kyegomez/swarms | `from swarms import` | 616 | 127 | 9 |
| MervinPraison/PraisonAI | `from praisonaiagents import` | 636 | 42 | 1 |
| livekit/agents | `from livekit.agents` | 588 | 492 | 19 |
| AgentOps-AI/agentops | `from agentops` | 528 | 126 | 14 |
| SylphAI-Inc/AdalFlow | `from adalflow.` | 520 | 25 | 3 |
| agentuniverse-ai/agentUniverse | `from agentuniverse.` | 502 | 4 | 0 |
| openlit/openlit | `import openlit` | 468 | 68 | 9 |
| InternLM/lagent | `from lagent.` | 454 | 106 | 12 |
| letta-ai/letta | `from letta_client` | 240 | 29 | 3 |
| lavague-ai/LaVague | `from lavague.` | 225 | 16 | 1 |
| superduper-io/superduper | `from superduper import` | 170 | 2 | 0 |
| Div99/agent-protocol | `from agent_protocol import` | 107 | 43 | 5 |
| awslabs/agent-squad | `from agent_squad.` | 104 | 6 | 0 |
| Upsonic/Upsonic | `from upsonic import` | 82 | 20 | 4 |
| raga-ai-hub/RagaAI-Catalyst | `from ragaai_catalyst` | 76 | 1 | 0 |
| melih-unsal/DemoGPT | `from demogpt_agenthub` | 27 | 0 | 0 |

- Similar to agent frameworks, we apply the contributor count filter and retain repositories with at least two contributors.
- To ensure the repositories are not one-time activities, we include repositories with a lifetime of at least one month. We measure the lifetime of a repository by calculating the difference between the last commit date and the creation date for the repository, following the previous studies in the testing domain (Hassan and Rahman, 2022)

Table 4: AI Agentic Applications: Filtering Toy Repositories

| **Filter** | **Repository Count** |
|---|---|
| Non-fork | 7,397 |
| Distinct | 7,074 |
| Contributor count $\geq 2$ | 1,957 |
| Lifetime $\geq 1$ month | 1,222 |
| Commit frequency $> 2$ per month | 1,144 |
| Number of test files $> 1$ | 439 |

- To ensure regular development activity, we include repositories with a commit frequency of at least two per month throughout their lifetime (Munaiah et al., 2017).
- Finally, we retain only repositories containing at least two test files, identified via filename prefixes (e.g., `test_*.py`).

For the filtering process, we first collect metadata using the GitHub API sequentially and store the metadata in a PostgreSQL database for each repository. We then apply the filtering conditions individually, ensuring each step refines the dataset for the following filtering condition.

Table 4 presents the repository count at each filtering step. From our initial 7,397 repositories, these filters remove 6,958 repositories, and we prepare the remaining list of 439 repositories for the next step.

#### *5.2.3 Mining and Deduplicating Test Functions*

To extract test functions from the 439 filtered agentic application repositories, we apply the same AST-based mining approach described in Section 5.1.4. During this process, we exclude files located under *site-packages* directories to avoid importing test functions originating from the agent frameworks.

However, upon inspection of the extracted data, we still observe duplicated test functions across different repositories. Many agentic applications have directly copied internal directories, including test files, from the frameworks they built upon, resulting in the same test function from the agent frameworks appearing in multiple agentic applications. To ensure analytical validity, we employed a SHA-256-based hashing technique (Gueron et al., 2011) to eliminate such duplicates.

A test function is considered a duplicate if its hash exactly matches a function hash from one of the agent frameworks it imported. The de-duplication process involves the following steps:

- For each test function extracted from an AI agent framework, we compute a SHA-256 hash (`HF`) based on its normalized function signature (including name and body). These are stored in Elasticsearch alongside the function metadata.
- For each test function extracted from an agentic application, we similarly compute a SHA-256 hash (`HA`).
- Given that an agentic application may depend on multiple frameworks, we compare each `HA` against all `HF` values from the relevant frameworks.
- If a match is found, the corresponding test function is marked as a duplicate and we exclude it.

- Otherwise, the function is retained and indexed as a unique test case.

This de-duplication procedure removes the redundant test functions, resulting in the final dataset of 110,891 unique test functions sourced from agentic applications, all stored in our Elasticsearch index for further analysis.

#### *5.2.4 Stratified Random Sampling*

Given the size of the agentic application dataset, e.g., 110,891 unique test functions, manually analyzing every function is impractical. Moreover, we observe that the distribution of test functions is highly skewed: the top 10 repositories alone account for approximately 70% of the total test functions. As with the framework repositories ( Section 5.1.5), applying simple random sampling in this context risks disproportionately sampling from larger repositories, thereby obscuring practices in smaller ones.

To mitigate this imbalance, we again apply *stratified random sampling*, treating each agentic application repository as a separate stratum and allocating sample size proportionally based on the number of test functions in each repository. Following the same statistical procedures used for the framework dataset, we target a 95% confidence level and a 5% margin of error. This results in a required sample size of 382 test functions, calculated using standard formulas for estimating proportions in finite populations. This approach ensures representative coverage of testing practices across agentic applications, while maintaining consistency with the sampling strategy used for framework repositories.

### 5.3 Card Sorting

After filtering and sampling, we obtain two representative datasets of test functions: one from AI agent frameworks and another from agentic applications. To investigate testing patterns within these datasets, we need to systematically analyze how test functions are structured, what they aim to validate, and which components of the agent systems they target.

However, as test functions are written in code and vary widely in style, naming, and structure, they represent a form of textual data that is not easily categorized through automated means. To address this, we employ qualitative card sorting techniques, which are widely used in empirical software engineering to extract themes and patterns from code artifacts (Spencer, 2009).

We first apply *hybrid card-sorting* to identify emergent testing patterns, i.e., recurring structures, strategies, or practices embedded in the test code. This inductive approach allows patterns to emerge organically from the data while also allowing a set of predefined categories. (Zampetti et al., 2022). Next, to identify which parts of the agent architecture are being tested, we extract the *subject under test* (SUT) and apply a *closed card sorting* Zhao et al. (2024) to map those SUTs to the core agentic components defined in Section 3.

The following subsections describe these three-step procedures: hybrid card-sorting for testing patterns, extraction of SUTs, and closed card sorting for mapping SUTs to components.

### *5.3.1 Hybrid Card Sorting for Testing Patterns*

To analyze testing patterns in test functions extracted from AI agent frameworks and agentic applications, we apply hybrid card-sorting, following methodologies established in prior empirical studies (Zampetti et al., 2022). In a hybrid card-sorting, elements of both open and closed card sorting are combined, e.g., participants are given predefined categories into which to sort content (as in closed sorting), but are also free to create new categories when existing ones are insufficient (as in open sorting). Before detailing our card-sorting procedure, we first define what constitutes a testing pattern in this context and describe the underlying structure of unit tests.

In unit testing, a test function typically follows the AAA (Arrange-Act-Assert) structure (Wei et al., 2025). First, a test fixture is initialized to establish the desired state (`Arrange`); next, a command is executed on the subject under test (`Act`); and finally, the output is verified using assertions (`Assert`) (Van Rompaey and Demeyer, 2008; Tao, 2009). Among these stages, the `Act` phase tends to be straightforward and consistent, e.g., involving a direct invocation of a SUT function. In contrast, the `Arrange` and `Assert` phases often vary significantly depending on developer strategy, testing framework, and system complexity.

*Testing patterns* refer to these recurring strategies for structuring setup and verification logic in test functions. They improve the maintainability, diagnosability, and reliability of tests (Gonzalez et al., 2017; Meszaros et al., 2003). Practitioners may use mocking, patching, parameterization, or custom assertions to isolate behavior, simulate dependencies, or evaluate correctness.

To illustrate how practitioners set up a test function and verify the output of the SUT, we present a sample unit test function in Figure 2 that validates the behavior of the `oas3_openai_text_to_embedding` function. This function internally interacts with the OpenAI API, and the test isolates it by mocking external dependencies. Specifically, the test uses the `patching` technique to intercept and override the behavior of the `aiohttp.ClientSession.post` method (lines 3-4). The mocked response is configured to return a predefined JSON payload, simulating a successful API call (lines 5-8). This setup phase demonstrates two common structural patterns, i.e., `Mocking` and `Patching` to handle the dependencies. Additionally, the test includes verification logic that checks both the existence and structure of the returned embedding (lines 14-17). These lines reflect a combination of basic presence assertions (e.g., `assert result`) and threshold assertions (e.g., `assert len(result.data[0].embedding) > 0`), which are often used to confirm the correctness and completeness of test outputs.

For card sorting, the first and fourth authors of this study acted as independent raters. The first author has 10 years of industry experience and 3 years of research experience, and the fourth author has 12 years of industry and 9 years of research experience. Prior to initiating the card-sorting, all raters reviewed five foundational studies on testing in software engineering and machine learning (Zhu et al., 2025; Zamprogno et al., 2022; Openja et al., 2024; Gonzalez et al., 2017; Zhang and Mesbah, 2015). These studies provided representative examples of known patterns but were not treated as a closed card-sorting framework, meaning raters were encouraged to identify novel patterns that may emerge from AI agent test functions.

At the beginning, we randomly chose 15% of the sampled test functions from both agent frameworks and agent applications (58 test functions from each), which were independently sorted by the first and fourth authors. Each rater spent approximately

```python
async def test_embedding(mocker):
    config = Config.default()
    mock_post = mocker.patch('aiohttp.ClientSession.post')
    mock_response = mocker.AsyncMock()
    mock_response.status = 200
    data = await aread(Path(__file__).parent /
    ↪  '../../data/openai/embedding.json')
    mock_response.json.return_value = json.loads(data)
    mock_post.return_value.__aenter__.return_value = mock_response
    type(config.get_openai_llm()).proxy =
    ↪  mocker.PropertyMock(return_value='http://mock.proxy')
    llm_config = config.get_openai_llm()
    assert llm_config
    assert llm_config.proxy
    result = await oas3_openai_text_to_embedding('Panda emoji',
    ↪  openai_api_key=llm_config.api_key, proxy=llm_config.proxy)
    assert result
    assert result.model
    assert len(result.data) > 0
    assert len(result.data[0].embedding) > 0
```

Arrange

Act

Assert

Fig. 2: A Sample Test Function marked with Arrange, Act, Assert blocks.

20 hours uncovering the testing patterns from the sampled functions. Each rater recorded their findings in a structured spreadsheet containing:

- Repository name
- Test function name
- Full test function source code
- Observed testing patterns (arrange and assertion level)

Because a single test function may exhibit multiple patterns, multi-label annotations were adopted. To ensure contextual understanding, raters consulted test code, SUT code, and README files. They also cross-referenced candidate patterns with those discussed in the five pre-reviewed studies when evaluating similarity or novelty.

After completing this sorting, the raters held a dedicated meeting to align their taxonomies. Importantly, this discussion focused exclusively on the naming and granularity of pattern categories and not on individual card-sorting results to avoid bias. During this calibration, the raters identified and resolved three types of discrepancies:

(i) Inconsistent naming of equivalent patterns, leading to redundant entries.
(ii) Overly broad pattern categories that conflated multiple strategies.
(iii) Missing distinctions where raters overlooked subtle but meaningful pattern variations.

To address these issues, the raters developed a consensus taxonomy by merging redundancies, refining naming conventions in alignment with existing literature, and agreeing on consistent granularity levels. Once the taxonomy was finalized, the raters re-sorted the 15% dataset. Given the multi-label nature of this task, we measured inter-rater agreement using Jaccard similarity, following prior work on similar classification efforts (Parker et al., 2024). The agreement scores were 0.88 for agent framework test patterns and 0.85 for agentic application patterns, indicating high reliability of the coding scheme. These scores exceed the commonly accepted threshold of 0.80 for strong agreement, suggesting that the labels are well-defined and

consistently interpretable. Based on this high level of agreement, the remainder of the dataset was coded by the first author using the finalized taxonomy.

#### *5.3.2 Extracting SUTs*

To understand which parts of the agent architecture are being tested, we first identified the *Subject Under Test* (SUT) in each sampled test function. The SUT refers to the primary object, function, or module that the test function aims to execute and validate (Van Rompaey and Demeyer, 2008). While every test function must target at least one SUT, extracting this information is non-trivial, as test code often lacks explicit references to the name of the SUT.

Similar to the process used for identifying testing patterns, we sampled 15% of the test functions from both the agent framework and agentic application datasets for component-level labeling. Two authors (first and fourth) independently served as raters and categorized each test function by recording the repository name, function name, and its corresponding System Under Test (SUT) in a spreadsheet. For instance, in the test function shown in Figure 2, the primary method under test is `oas3_openai_text_to_embedding`, which generates embeddings from input text. An embedding is a representation of an input text in a numerical format, typically as vectors or matrices in a high-dimensional space (Patil et al., 2023). Based on this functional purpose and domain context, we labeled the SUT as `Embedding`.

After completing the initial labeling round, the raters compared their annotations and assessed inter-rater reliability using Cohen's Kappa (Landis and Koch, 1977), as each test function corresponds to a single SUT. The resulting agreement score was 0.52, indicating moderate agreement and highlighting the presence of labeling conflicts. To resolve these conflicts, a third rater, the second author of the paper, was introduced as an adjudicator. Following conflict resolution, the agreement score improved to 0.87, demonstrating strong consistency in the finalized taxonomy. Based on this high agreement, the first author completed the remaining labeling using the consensus definitions.

#### *5.3.3 Closed Card Sorting for Components*

With the SUTs identified in the previous step (Section 5.3.2), we performed a closed card-sorting to uncover the higher-level architectural components of agent frameworks and applications being tested. We considered the components described in Section 3.3 as the source of components and grounded each SUT to the relevant component through the closed card sorting process.

Following the methodology outlined in Section 5.3.1, the first and second authors jointly map the SUTs to a component label. For example, in Figure 2 , the extracted SUT is an embedding created and used for various purposes, e.g., similarity search, context retrieval, knowledge storage, agent frameworks, and agentic applications. As defined in Section 3.3, embeddings fall under `Resource Artifacts`.

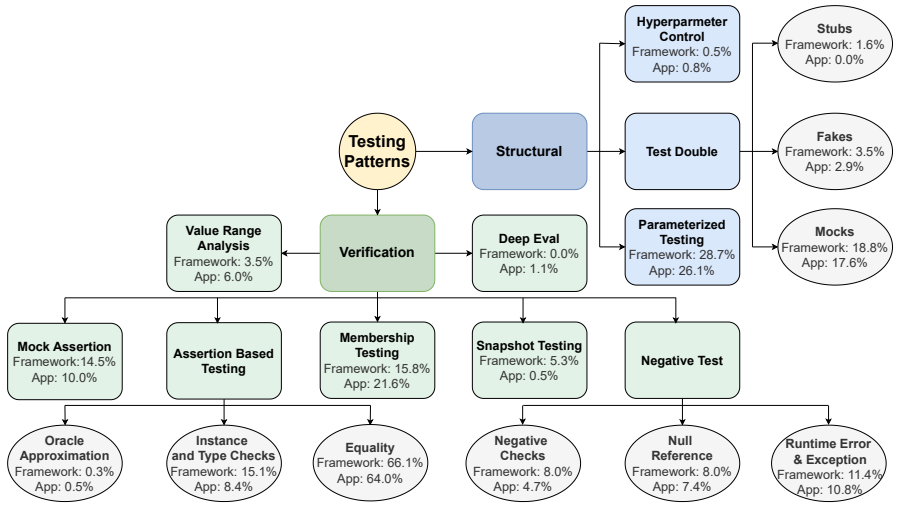



Fig. 3: Overview of testing patterns observed in agent frameworks and applications. The three structural patterns (highlighted in blue) and seven verification patterns (highlighted in green) are organized under the top-level testing patterns. Elliptical nodes represent sub-patterns grouped under their corresponding high-level categories.

## 6 RQ1: What testing practices are commonly adopted by practitioners to evaluate open-source AI agent frameworks and agentic applications?

### 6.1 Motivation

As established in the Introduction, the prevailing benchmark-driven evaluation culture often fails to capture the practical reliability of agentic systems, overlooking critical failure modes like prompt decay (Bhargava et al., 2024) and tool-use errors that are already documented in the field (Ma et al., 2024; Kokane et al., 2025). While software testing offers a path to bridge this reliability gap, the inherent non-determinism and non-reproducibility of foundation models (FMs) fundamentally challenge the application of traditional, deterministic testing techniques (Hassan et al., 2024).

This conflict creates a crucial knowledge vacuum: with no established best practices for this new paradigm, it remains unknown how practitioners are adapting their testing strategies to cope with these challenges. Therefore, to bridge this gap, we conduct the first large-scale empirical investigation to systematically identify and catalog the testing patterns that have emerged in practice. We specifically analyze how developers structure their tests to manage variability (the Arrange phase of a test) and what strategies they employ to verify unpredictable yet potentially correct outputs (the Assert phase), as more than 75% of tests follow the AAA pattern in recent times (Wei et al., 2025) in traditional software engineering. The goal is to produce a foundational catalog of these real-world strategies, providing the first empirical baseline on how the open-source community is negotiating the unique quality assurance hurdles of the agentic paradigm.

### 6.2 Approach

To identify testing patterns in AI agent frameworks and agentic applications, we perform `hybrid card-sorting` on test functions mined from open-source repositories, as described in Section 5.3.1. Hybrid card-sorting allows us to systematically extract recurring testing patterns related to test structure, e.g., how to `Arrange` the test function and verification strategies and how to `Assert`. This process not only focuses on the existing patterns but also enables uncovering emergent patterns that have not been seen in the literature before. We provide an example of each pattern in Appendix A.

### 6.3 Findings

**We identify three structural patterns for "arranging" test functions and seven verification patterns for "asserting" test outcomes**. These patterns, uncovered through the hybrid card-sorting process, along with their frequencies in both agent frameworks and agentic applications are illustrated in Figure 3. Additionally, we discovered three sub-patterns under the *Test Double* structural pattern, and three sub-patterns each under the verification patterns *Assertion Based Testing* and *Negative Test*.

**Despite the novelty of AI agent frameworks and agentic applications, 80% of their testing patterns are directly inherited from classical software engineering or ML applications, reaffirming the robustness of classical testing strategies.** Table 5 summarizes the testing patterns we observed and their corresponding origins. We find that 8 out of 10 patterns have been previously reported in either ML application testing (Openja et al., 2024), or in general software engineering literature (Zhu et al., 2025; Zamprogno et al., 2022; Fujita et al., 2023; Lam et al., 2018), or in both. While ML testing research has focused more on verification patterns than on structural patterns, prior studies have still documented the use of structural patterns, e.g., test doubles (Wan et al., 2019) in ML applications. Overall, our findings suggest that practitioners are not reinventing the wheel but are strategically adapting battle-tested patterns to a new domain. The key shift is not in the patterns themselves, but in their frequency and combination.

**To evaluate the non-deterministic response of underlying FM in agentic applications, practitioners leverage a new state-of-the-art verification pattern, i.e., *DeepEval*.** In general, DeepEval integrates multiple evaluation techniques, including *G-Eval* (Liu et al., 2023b) for assessing answer relevancy, task success, and hallucination detection, and *RAGAS* (Es et al., 2024) for evaluating faithfulness and precision. Both approaches leverage the *LLM-as-a-judge* paradigm (Zheng et al., 2023), in which a language model interprets test outputs using a set of criteria generated through *Chain-of-Thought prompting* (Wei et al., 2022b). Additionally, *DeepEval* supports user-defined logic through a rule composition mechanism based on directed acyclic graphs known as *DAGMetric* (Confident, 2024).

We illustrate a test function that uses *DeepEval* in Figure 4 to explain how *DeepEval* helps agentic applications to tackle the uncertainty from FM. In this example, `test_relevant_content_retrieved` leverages *DeepEval* to evaluate a resume

Table 5: Distribution of testing patterns across agent frameworks, agentic applications, machine learning (ML) applications, and traditional (trad.) software, expressed as percentages and sorted alphabetically by pattern name. Green-highlighted cells denote patterns newly identified in agent-based systems that were not previously reported in either ML or traditional software domains. "NS" indicates the pattern was not studied in the corresponding domain. ML application data (marked with blue superscript[1]) is from (Openja et al., 2024); traditional software data is sourced from (Lam et al., 2018)[2], (Zhu et al., 2025)[3], (Zamprogno et al., 2022)[4], and (Fujita et al., 2023)[5].

| Testing type | Testing patterns | % Agent framework | % Agentic apps | % ML apps[1] | % Trad. software |
|---|---|---|---|---|---|
| Structural | Hyperparameter Control | 0.5 | 0.8 | 0 | 0 |
| | Parameterized Testing | 28.7 | 26.1 | NS | 9.0[2] |
| | Test Double | 23.9 | 20.5 | NS | 14.7[3] |
| Verification | Assertion Based Testing | 81.5 | 72.9 | 24.9 | 56.9[4] |
| | DeepEval | 0.0 | 1.1 | 0.0 | 0.0 |
| | Membership Testing | 15.9 | 21.6 | 8.8 | 13.8[4] |
| | Mock Assertion | 14.5 | 10.0 | 1.0 | 4.0[4] |
| | Negative Test | 27.4 | 22.9 | 24.5 | 8.1[4] |
| | Snapshot Testing | 5.3 | 0.5 | 0.0 | 7.0[5] |
| | Value Range Analysis | 3.5 | 6.0 | 9.2 | 2.2[4] |

retriever agent's output against a job description and target company. The test function invokes the G-Eval metric, which operationalizes the LLM-as-a-judge to handle non-deterministic responses. The user-provided evaluation_steps (line 18) serve as explicit evaluation criteria, instructing the judge to assess the answer's relevancy by verifying that the correct company name is included. This approach validates the semantic correctness of the agent's output, with the test passing if the LLM-as-a-judge's confidence score meets the specified 0.7 threshold.

**To achieve test reproducibility, practitioners employ a hyperparameter control mechanism as a structural pattern while testing agent frameworks.** We illustrate a real-world test function in Figure 5 where temperature, a hyperparameter through which randomness of FM's output can be controlled, is set as 0 (line 6). In general, a higher temperature during generation allows FM to explore more and produce more diverse outputs (Xu et al., 2022). However, for the test function where practitioners need reproducible output every time, they set the temperature to 0 or a very low value.

**Adoption of emerging patterns like *DeepEval* and *Hyperparameter Control* is negligible (around 1%), revealing a significant gap between state-of-the-art techniques and practitioner awareness.** Figure 3 shows that a mere 1.1% of test functions of agentic applications leverage DeepEval to validate foundation model outputs, while just 0.5% and 0.8% of test functions from agent frameworks and agentic applications utilize hyperparameter control mechanisms,

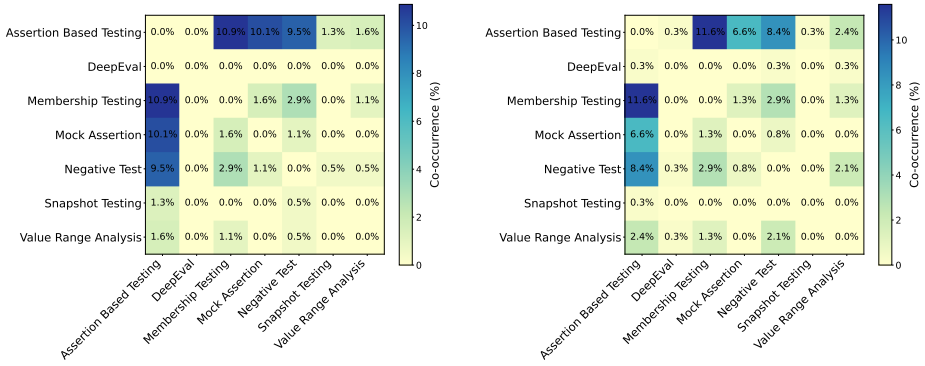
```
1  def test_relevant_content_retrieved(user_proxy: UserProxyAgent,
   ↪  resume_retriever_agent: ResumeRetriever, job_description: str, company: str)
   ↪  -> None:
2      message = "Here is the job description: " + job_description
3      chat_outcome = get_chat_outcome(user_proxy, resume_retriever_agent, message)
4      assert_test(
5          LLMTestCase(
6              input=message,
7              actual_output=chat_outcome,
8              context=[company]
9          ),
10         [
11             GEval(
12                 name="Inclusion",
13                 evaluation_params=[
14                     LLMTestCaseParams.INPUT,
15                     LLMTestCaseParams.ACTUAL_OUTPUT,
16                 ],
17                 threshold=0.7,
18                 evaluation_steps=[
19                     f"Check that the output contains the company name
                       ↪  '{company}'",
20                     f"Check that the output does not contain any company name
                       ↪  other than '{company}'",
21                 ],
22             )
23         ],
24     )
```

Fig. 4: Example DeepEval test case that verifies whether the retrieved output includes only the correct company name. Pattern starts by triggering *assert_test* on line 4, and GEval is invoked on line 11. Evaluation parameters are configured in lines 14–15, while the threshold and validation steps are defined in lines 17–20.

```
1  def test_10_concurrent_API_calls(self):
2      tools = []
3      with open('./data/schemas/get-headers-params.json', 'r') as f:
4          tools = ToolFactory.from_openapi_schema(f.read(), {})
5      ceo = Agent(name='CEO', tools=tools, instructions="You are an agent that
       ↪  tests concurrent API calls. You must say 'success' if the output contains
       ↪  headers, and 'error' if it does not and **nothing else**.")
6      agency = Agency([ceo], temperature=0)
7      result = agency.get_completion("Please call PrintHeaders tool TWICE at the
       ↪  same time in a single message. If any of the function outputs do not
       ↪  contains headers, please say 'error'.")
8      self.assertTrue(result.lower().count('error') == 0,
       ↪  agency.main_thread.thread_url)
```

Fig. 5: Test function where temperature is set as 0 for forcing FMs to always select the most probable next token during text generation, eliminating any randomness in the output.

respectively, during test setup. This low adoption rate likely reflects their recent introduction, lack of awareness, or the associated steep learning curve to use these patterns.




(a) Agent Framework (b) Agentic Application

Fig. 6: Co-occurrence frequency of verification patterns in the same test function in agent framework and agentic application.

**Parameterized testing is the most heavily adopted structural pattern to handle input variability in agent-based systems, being nearly three times more frequent than in traditional software.** As shown in Table 5, this pattern appears in 28.7% of framework tests and 26.1% of application tests. This widespread use of executing a single test with multiple input values (Tillmann and Schulte, 2005) starkly contrasts with the 9% adoption rate previously reported in traditional software engineering (Lam et al., 2018). This pronounced reliance on parameterization suggests that developers of agentic systems require scalable testing practices to validate logic against the wide range of dynamic data and probabilistic outputs inherent to systems driven by foundation models.

**While strict assertion-based testing remains the primary verification method (72–82%), its frequent co-occurrence with more flexible patterns such as membership testing, mock assertion, and negative testing signals a strategic adaptation to the inherent uncertainty in agent frameworks and agentic applications**. As shown in Table 5, assertion-based testing is employed 3–4 times more frequently than the next most common pattern. Yet, as the co-occurrence matrix in Figure 6 reveals, assertion-based tests are often augmented with other verification patterns. In agent frameworks, 40.2% of test functions using assertion-based verification also include at least one flexible verification strategy, compared to 36.7% in agentic applications. These complementary patterns offer more adaptable validation: membership testing checks for the presence of a pattern in the output (Openja et al., 2024), mock assertions verify interactions with test doubles, e.g., mock objects (Zhu et al., 2025), and negative testing ensures robust error handling (Openja et al., 2024). This trend highlights a necessary evolution in verification, as the probabilistic and semantically variable responses from FM-based agents demand approaches that balance traditional rigor with the flexibility needed to validate correctness in the face of uncertainty.

**Summary of RQ1**

- While practitioners predominantly rely on traditional testing patterns to accommodate the challenges of non-determinism and non-reproducibility in agentic systems, specialized patterns (i.e., *DeepEval* and *Hyperparameter Control*) remain rarely adopted despite being specifically designed to address these issues.
- This limited adoption might highlight either a lack of awareness within the developer community, the steep learning curve associated with these novel patterns, or insufficient integration with existing testing frameworks and tools.

## 7 RQ2: How do these testing practices map to the architectural components of the agent frameworks and agentic applications?

### 7.1 Motivation

A major challenge in studying agent architectures from a testing perspective is the ecosystem's rapid and continuous evolution. Since the inception of our study in September 2024, several major updates have already reshaped the landscape: the introduction of the Model Context Protocol (MCP) for tool use in November 2024 (Hasan et al., 2025), new agent-to-agent communication primitives (A2A) in April 2025, and formal agent communication and networking protocols (ACP and ANP) in May 2025 (Ehtesham et al., 2025). To ensure our analysis is not tied to these ever evolving components and remains durable over time, we map the concrete Systems Under Test (SUTs), i.e., the specific features or modules tested in real-world agent frameworks and agentic applications, to a stable, canonical architectural component derived from established agent literature as described in Section 3.3. This canonical model enables us to systematically cover the full spectrum of architectural concerns in existing agentic systems. It also keeps our methodology extensible. For instance, as A2A emerges as a new technique (e.g., for inter-agent messaging), components using A2A can be mapped onto an existing canonical component (such as `Communication Artifact`).

Understanding how different components of the agent ecosystem are tested is critical for identifying whether current testing strategies adequately address component-specific risks and behaviors. For instance, `Boundary Artifacts` interact with external systems, which can be unreliable. This necessitates the use of testing patterns that can isolate unreliable components. On the other hand, `Constitutive Entities` regulate the behavior of the flow and raise exceptions when regulations are breached. Consequently, testing patterns that focus on exception handling are more relevant for constitutive entities. While RQ1 has already uncovered the types of testing patterns practitioners adopt, from novel approaches like *DeepEval* to the strategic adaptation of traditional less strict methods, what remains unclear is how these patterns are applied across different architectural components. Therefore, we hypothesize that testing effort is not uniform; practitioners strategically apply more intensive or specialized testing to components they perceive as riskier. By mapping the testing patterns from RQ1 to specific architectural components, we will validate this hypothesis and, more importantly, identify critical components that may be potentially under-tested.

## 7.2 Approach

We begin by manually identifying the subjects under test (SUTs) from each test function, as outlined in Section 5.3.2. These SUTs are then mapped to canonical agent architectural components using a closed card-sorting process based on the taxonomy described in Section 5.3.3. Finally, we analyze the co-occurrence of testing patterns (Table 5) within these canonical components to determine which testing strategies are most commonly applied to which parts of the system.

## 7.3 Findings

In this subsection, we first define all the SUTs observed in the sampled test functions during our manual review process and map them to the canonical components of the JaCaMo framework. Next, we present the prevalence of these components in the test functions. Finally, we report the component-wise testing patterns, including those defined in Table 5.

### *7.3.1 Component Mapping With SUT*

**We found that a total of 35 different SUTs are being tested, and mapped these SUTs to 13 canonical agent components**. An overview of the mapping is illustrated in Figure 7. We identify at least one and a maximum of seven SUTs that fall under different architectural components. Table 6 provides a high-level description of the SUTs under each component.

Table 6: Summary of System Under Test (SUT) Components and Characteristics ordered in the same order the components are described in Section 3.3

| Component | SUT | Characteristics |
|---|---|---|
| Belief Base | Memory | Persistent storage of trial history, observations, and learned knowledge (Zhang et al., 2024). |
| | Cache | Short-term semantic cache that holds recently accessed results for reuse (Bang, 2023; Zhang et al., 2025). |
| | Filestore & Document Store | Structured or unstructured document repositories used for long-term reference. |
| | Knowledge Base | Runtime-accessible structured knowledge repositories integrated into the agent's memory framework. |
| Trigger | Prompt | Text-based inputs that provide instructions to foundation models (FMs) for generating outputs (Marvin et al., 2023). |

| Component | SUT | Characteristics |
|---|---|---|
| Context | RAG | Agents supply domain-specific context to FMs via retrieval-augmented generation (RAG), improving relevance and factuality (Liu et al., 2025). |
| | VectorDB | Vector databases store contextual and domain knowledge for retrieval during task execution (Barron et al., 2024). |
| | Context | Some test cases provide context through plain text, such as conversation history, apart from formal RAG and vector-based storage. |
| Plan Body | Foundation Models | Agents use FMs (e.g., LLMs) to interpret goals, reason, and generate actionable plans. Chain-of-thought reasoning enables structured multi-step task execution (Liu et al., 2025; Shen et al., 2024; Wei et al., 2022b). |
| Communicative Action | Messages & Events | Agents receive and dispatch messages or events to collaborate with other agents and systems (Nascimento et al., 2023). |
| Resource Artifacts | Internal Tools | Shared resources that different agents can (re)use to solve common problems, such as document managers, calculators, and command-line tools. |
| | MCP | Model Context Protocol (MCP) enables standardized discovery and invocation of tools via external MCP servers (Hasan et al., 2025). |
| | Ranker | Tools that rank or score retrieved results to improve the accuracy of the responses and decisions (Liu et al., 2025). |
| | Evaluators & Validators | Modules that evaluate and validate the outputs, prompts, or intermediate artifacts generated by agents or tools. |
| | Tokenizer | Converts strings into token sequences (IDs) used internally by FMs (Rajaraman et al., 2024) and also used for knowledge and retrieving context. |
| | Embeddings | Semantic vector representations of various inputs (e.g., text, images) used for retrieval and reasoning (Liu et al., 2025). |
| | Databases | Data repositories used for storing structured information during agent execution. |
| | Data Processor | Serializers, deserializers, parse, or transformers used to prepare or normalize input/output data formats. |

| Component | SUT | Characteristics |
|---|---|---|
| Coordination Artifacts | Workflow | Defined sequences of actions including perception, planning, action, and adaptation, guiding agent behavior (Li et al., 2024). |
| Boundary Artifacts | External Connectors | Interfaces for invoking external services or tools as part of the agent workflow (Liu et al., 2025). |
| Registry | Tool Registry | Central repository for registering, discovering, and invoking tools and agents; one test function also involved a model registry (Liu et al., 2025). |
| Observable Property | Loggers | Components that collect telemetry and debugging data from agent executions (Hassan et al., 2024). |
| | Watchers | Monitors that detect changes in files, databases, or environments and notify agents accordingly. |
| | Metrics | Quantitative indicators used to evaluate agent behavior, accuracy, or performance (Liu et al., 2025). |
| | Output | Results produced by agents, such as code, text, reports, or files. |
| Role | Roles | Assigned responsibilities or personas that guide agent behavior, tool selection, and orchestration (Liu et al., 2025). |
| Groups | Agency | Multiple agents with different roles can collaborate to form an agency. |
| Constitutive Entity | Rate Limiter | Mechanism that constrains frequency of resource or API access (e.g., FM calls). |
| | Guardrails | Constraints on inputs/outputs to ensure safety, alignment, and compliance (Liu et al., 2025). |
| | Timeout | Fail-safes that interrupt long-running or unresponsive tasks. |
| | Security | Authentication and authorization controls for safe agent-environment interactions. |
| | Infrastructure Hosting | Deployment environments (e.g., Docker, VMs) that host agent services. |
| | Configuration | Schema-driven configuration files used to tune agent behavior. |
| | Concurrency | Controls limiting the number of simultaneous actions or requests made by an agent. |

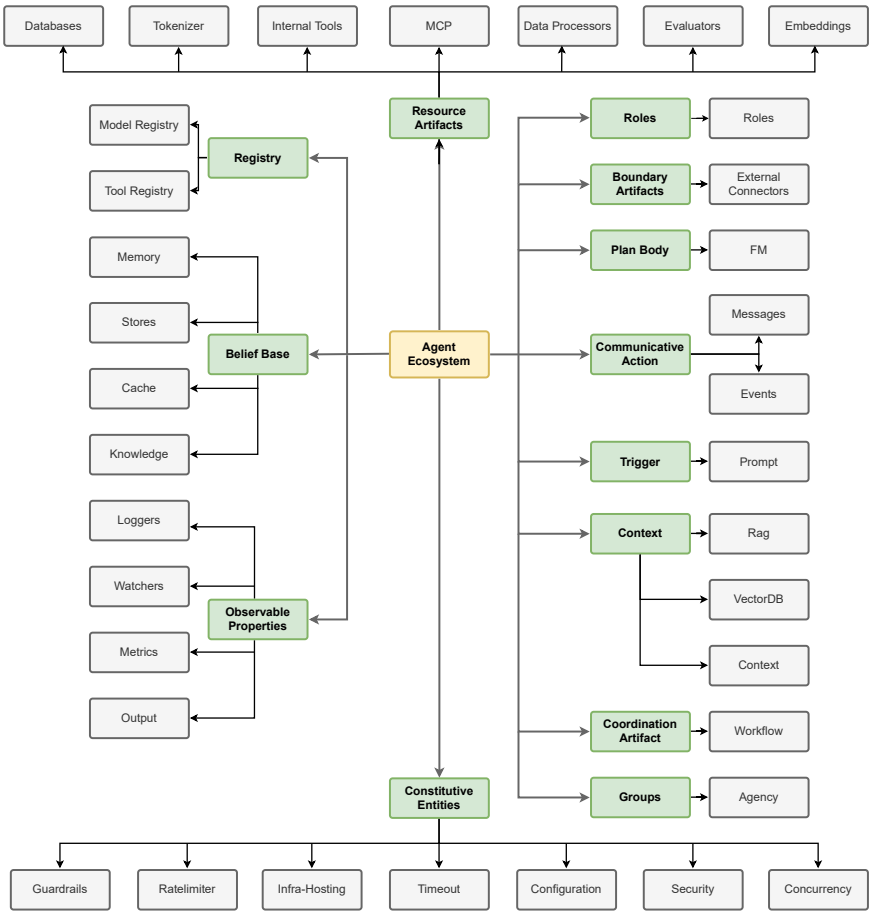



Fig. 7: An overview illustrating the mapping between SUTs and canonical agent architectural components extracted from the sample dataset. The green nodes represent architectural components derived from existing literature, while the gray leaf nodes correspond to specific SUTs identified from the sampled test functions.

### *7.3.2 Component-wise Test Frequency*

**AI agent testing inverts the traditional ML paradigm, shifting testing effort from the non-deterministic model to the deterministic artifacts around it**. Figure 8 presents the distribution of test functions across architectural components within agent frameworks and agentic applications. Our analysis shows where the testing effort is invested: `Resource Artifacts` receive the lion's share of testing effort, accounting for 29.7% of tests in frameworks and 39.2% in applications. This is in stark contrast to the `Plan Body` (containing the FM), which receives much lower direct testing effort (<= 5% in both cases). The disparity highlights a dramatic reallocation of engineering effort compared to traditional ML, where model testing

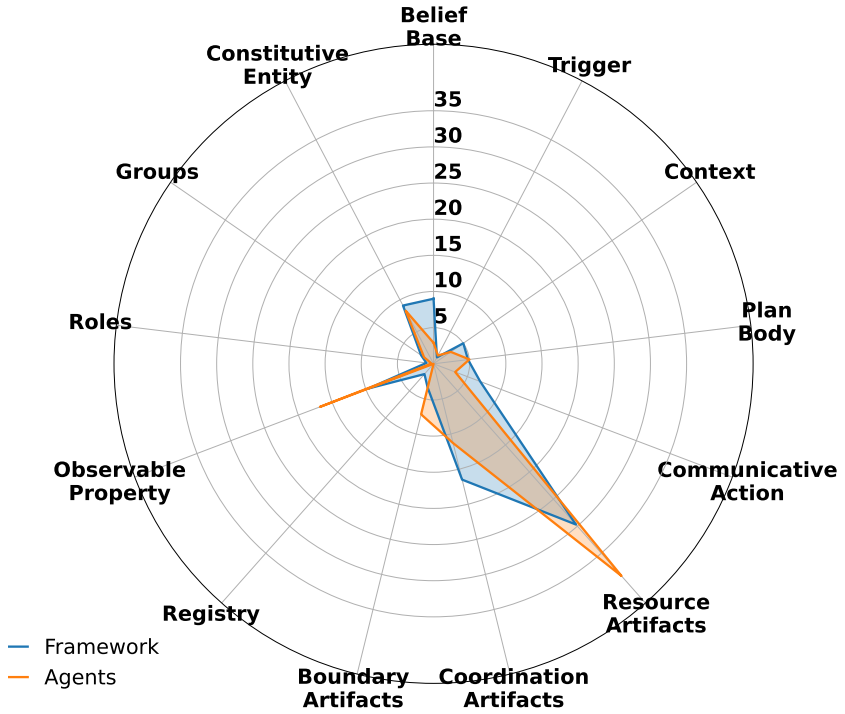



Fig. 8: Components tested in Agent Ecosystem

typically commands 24–30% of the focus (Openja et al., 2024). This suggests that developers are strategically investing their limited testing resources in the components they can reliably control and verify.

**Despite the Foundational Model's strong dependency on `Triggers` (prompts), testing this component remains critically under-addressed, highlighting a substantial blind spot in ensuring the robustness of agents frameworks and agentic applications**. As shown in Figure 8, around 1% of test functions across both frameworks and applications are dedicated to the `Trigger` component. This low investment is deeply concerning, as prior work has demonstrated that prompt quality, e.g., example selection, ordering, and contextual richness, significantly affects FM performance (Wu et al., 2024b). Moreover, prompt brittleness poses a growing risk as remote foundation models continue to evolve, often breaking existing prompts and degrading agent behavior unless regression testing is in place (Ma et al., 2024). The near-total absence of prompt regression testing leaves agentic systems highly vulnerable to unexpected decay in functionality, highlighting an urgent need for systematic prompt validation.

**The top four architectural components tested in both agent frameworks and agentic applications are identical, suggesting potential over-**


Fig. 9: Component-wise Testing Patterns

**lapping of testing effort across development layers**. As shown in Figure 8, the most frequently tested components in agent frameworks are `Resource Artifacts`, `Coordination Artifacts`, `Observable Property`, and `Constitutive Entity`, which together account for 65% of test functions. Interestingly, these same four components dominate in agentic applications as well, comprising 75.6% of test functions. This overlap indicates that both framework and application developers are heavily investing in testing similar subsystems, potentially leading to redundant validation across layers. In contrast, other critical components receive limited attention, e.g., `Context`, which grounds agent behavior through retrieved knowledge, and `Group`, which enables multi-agent collaboration, are seldom tested. This uneven distribution of testing effort suggests a need for further investigation into how redundancy can be minimized and whether resources should be reallocated to under-tested yet crucial components in the agent ecosystem.

### *7.3.3 Component Wise Testing Patterns*

***Assertion-Based Testing* and *Parameterized Testing* are consistently used across all components in both agent frameworks and agentic applications, highlighting their flexibility and strong practitioner preference**. As shown in Figure 9, these two patterns show consistently high prevalence across the entire component landscape, unlike other testing patterns that show more component-specific adoption. This universal applicability stems from their fundamental utility, e.g., parameterized tests efficiently cover diverse inputs with minimal code, while assertion-based tests provide straightforward, deterministic checks. Both are well-established techniques in traditional software engineering, as discussed in Section 6, and their widespread usage in the agent ecosystem suggests that practitioners favor these familiar, battle-tested strategies when adapting to the complexities of FM-based agent systems.

**Although agent frameworks and agentic applications test overlapping architectural components, they adopt distinctly different testing patterns for 69% of the canonical components, revealing divergent testing priorities and practices**. Figure 9 shows that this divergence is prominent in components `Belief Base`, `Boundary Artifacts`, `Communicative Action`, `Constitutive Entity`, `Coordination Artifacts`, `Groups`, `Observable Property`, `Registry`, and `Roles`. The fundamental split in testing philosophy separates agent frameworks and applications: frameworks are tested for universal robustness, while applications are tested for specific, contextual correctness.

**For `Coordination Artifacts`, this means that frameworks prioritize validating the entire workflow state while applications focus on verifying final outcomes**. While both employ *Test Double* extensively, agent frameworks show stronger adoption of *Negative Testing*, *Snapshot Assertion*, and *Mock Assertion*, whereas agentic applications do not use *Snapshot Assertion* at all and instead rely on *Membership Testing* approximately 2.5 times more frequently. This contrast suggests that frameworks emphasize preserving workflow integrity by capturing intermediate states, whereas applications prioritize checking artifact presence with more flexible, less rigid criteria.

**This diverging behavior continues with `Constitutive Entities`, where framework testing aims for comprehensive edge-case coverage, while application testing validates specific, isolated interactions**. Agent frameworks more frequently use *Parameterized Testing*, *Negative Testing*, *Membership Testing*, and *Mock Assertion*, while agentic applications employ *Parameterized Testing* and *Mock Assertion* at 2.2x and 3.5x lower rates, respectively, but use *Test Double* 1.4x more often. This reflects a difference in emphasis: frameworks favor systematic exploration of edge-case behaviors in environment-defining components, whereas applications focus on simulating and verifying isolated behaviors through mocking and test doubles.

**This contrast is further evident in testing `Boundary Artifacts`, where the use of *Assertion-Based Testing* drops from 84.6% in frameworks to 55.6% in applications, while *Membership Testing* increases nearly 2.9x in applications.** While both employ *Test Double* to isolate third-party integrations, frameworks lean toward rigorous validation via *Mock Assertion* and *Snapshot Testing*, whereas agentic applications prefer lightweight, targeted checks tailored to their specific operational contexts.

**The novel patterns *DeepEval* and *Hyperparameter Control* are each confined to just one or two components, explaining how their current use is highly specialized and not yet generalized across the agent ecosystem**. The heatmap in Figure 9 shows that these patterns have found their initial niche use-cases. *DeepEval*, for instance, is used only in agentic applications for testing `Observable Property` and `Resource Artifacts`. This is its ideal niche: validating ambiguous, FM-generated artifacts, e.g., output like code, summaries, where traditional assertions fail. Likewise, *Hyperparameter Control* has found its primary use case in testing `Roles` within frameworks, directly addressing the critical need for reproducible role-wise agent behavior during development and also in `Plan Body` for forcing FMs to produce consistent output. While not yet in broad use, their presence in these key areas shows that novel patterns are successfully solving problems that were previously intractable, signaling their potential to expand as the agent ecosystem matures.

**Summary of RQ2**

- Practitioners strategically allocate testing efforts, prioritizing the test of deterministic infrastructure components such as Resource and Coordination Artifacts rather than the testing of non-deterministic models like Plan Bodies, indicating an inversion from traditional ML practices.
- While agent frameworks and agentic applications test overlapping canonical components, their testing philosophies diverge across 69% of components: frameworks emphasize general robustness through rigorous checks, whereas applications focus on context-specific correctness using more adaptive and relaxed patterns.
- There are critical testing blind spots in the form of the near-total neglect of the `Trigger` (prompt) component, exposing agents to significant risks of performance decay and silent failures as underlying foundation models evolve.

## 8 Implications

Our empirical study reveals a rapidly evolving testing landscape for FM-based AI agent eco-systems. While practitioners are adapting established software engineering principles, significant gaps and strategic misalignments are evident. The findings from our investigation into testing patterns (RQ1) and component-level focus (RQ2) carry profound implications for the key stakeholders in this ecosystem. In this section, we discuss these implications for practitioners and researchers, outlining actionable pathways to foster a more mature, reliable, and secure agentic ecosystem.

### 8.1 Implications for Practitioners

Our findings offer a clear roadmap for both the developers of agent frameworks and the developers building applications upon them. The core message is a call for a more conscious and stratified approach to testing, acknowledging the unique challenges posed by FMs.

#### *8.1.1 For Agent Framework Developers*

**Framework developers should integrate advanced semantic verification capabilities, e.g., *DeepEval*, into their testing infrastructure to address the inherent non-determinism of foundation models**. Our results in Section 6 show that while assertion-based testing is dominant, it is fundamentally ill-equipped to handle the semantically variable yet valid outputs from agentic applications in terms of relevancy, accuracy, truthfulness and precision. This forces practitioners into a brittle testing paradigm. The emergence of patterns like *DeepEval*, despite its low adoption ($\approx 1\%$), offers a clear path forward. Frameworks should provide built-in support for such LLM-as-a-judge techniques (Zheng et al., 2023). By incorporating evaluators like G-Eval for relevancy and RAGAS for faithfulness (Liu et al., 2023b; Es et al., 2024), developers can offer users a robust mechanism to validate the intent and semantic correctness of an agent's output, rather than relying on fragile, exact-match string comparisons. This would shift the quality assurance burden from the application developer to the framework, significantly enhancing the reliability of the entire ecosystem.

**Framework developers should establish and promote a testing contract to delineate testing responsibilities between framework and application layers**. The substantial overlap in testing effort, as detailed in Section 7.3.2, suggests redundant work being carried out across the ecosystem. Framework developers can mitigate this by defining a clear testing contract. This contract would specify the robustness guarantees that a compliant framework must provide (e.g., "all Coordination Artifacts have been validated against concurrency issues"). In return, it would guide application developers to focus their efforts on their specific logic, trusting the framework's certified guarantees. By publishing these guidelines, creating best-practice documentation, and potentially even offering a "framework certification", framework developers can foster specialization, reduce redundant work, and optimize the allocation of testing effort across the entire ecosystem.

### *8.1.2 For Agentic Application Developers*

**Agentic application developers must establish systematic prompt regression suites to mitigate the risks of model evolution and prompt brittleness**. Our analysis in Section 7.3.2 exposed a critical blind spot: the `Trigger` component (i.e., the prompt) is tested in around 1% of cases. This is a ticking time bomb for system stability. Because foundation models are frequently updated by their providers, often without detailed information about version differences (Ajibode et al., 2025), prompts that worked perfectly yesterday may fail silently tomorrow (Ma et al., 2024). Agentic applications are uniquely positioned to address this by creating regression testing harnesses for prompts. These harnesses could maintain a curated set of "golden" prompts and corresponding semantic outputs, automatically validating them against new or updated FM versions. By treating prompts as first-class, testable artifacts, agentic application developers can provide a crucial layer of stability, safeguarding applications from unforeseen behavioral degradation and ensuring long-term dependability.

**Developers should adopt hyperparameter control as a fundamental debugging technique to isolate the subject under test from FM-induced non-reproducibility**. The negligible adoption of Hyperparameter Control (0.5% in frameworks, 0% in apps) shown in Section 6 indicates a missed opportunity. When a test fails, developers face a critical ambiguity: is the defect in their own code, or is it a result of the FM's random output? By setting hyperparameters like temperature to 0, developers can motivate deterministic outputs from the FM, effectively reducing the randomness to create a reproducible testing environment (Xu et al., 2022). This allows them to systematically debug their application's Coordination Artifacts (workflows) and Resource Artifacts (tools) without the confounding variable of FM unpredictability. Once the application logic is confirmed to be sound, the temperature can be increased to test the system's behavior under more realistic, non-deterministic conditions.

**Application developers should shift their testing focus towards business logic and integration points by relying on the robustness guarantees of the underlying framework**. Our analysis in Section 7.3.2 revealed a duplication of effort: both framework and application developers concentrate their testing on the same four components (`Resource Artifacts`, `Coordination Artifacts`, `Observable Property`, and `Constitutive Entity`). While seemingly diligent, this

can be inefficient at times. As shown in Section 7.3.3, frameworks test these components for general-purpose robustness (e.g., using Snapshot Testing on workflows), whereas applications test them for specific use cases (e.g., using Membership Testing on outputs). Application developers should offload the responsibility of foundational robustness to the framework. This allows them to redirect their limited testing resources to what truly matters at the application layer: the correctness of their custom tools, the reliability of their Boundary Artifacts (external connectors), and the integrity of their unique business workflows.

### 8.2 Implications for Researchers

**Researchers should conduct empirical studies to diagnose the barriers preventing the adoption of novel, agent-specific testing patterns**. The contrast between the potential of patterns like *DeepEval* and *Hyperparameter Control* and their near-zero adoption (Section 6) points to a significant gap between the state-of-the-art and the state-of-the-practice. Is this gap caused by a lack of awareness, tooling immaturity, perceived complexity, or a cultural adherence to traditional testing paradigms? Future research, employing qualitative methods like surveys and interviews with practitioners, could uncover these root causes. The resulting insights would be invaluable for designing more intuitive testing tools, developing targeted educational materials, and creating theories of technology adoption tailored to the rapidly evolving FM-based AI agent engineering landscape.

**Researchers should formalize a comprehensive testing methodology for agentic systems that accounts for their unique architectural and behavioral characteristics**. Our study revealed two foundational shifts in testing practice: the strategic adaptation of "less strict" patterns (RQ1) and the inversion of testing effort away from the non-deterministic model and towards the deterministic infrastructure (RQ2). These emergent practices currently lack a unifying theoretical framework. Researchers are positioned to develop a "Theory of Agent Testing" that formalizes these observations. Such a theory would provide prescriptive guidance on which testing patterns are most effective for which canonical components. For instance, using *Test Double* for `Boundary Artifacts`, *Negative Testing* for `Constitutive Entities`, and *DeepEval* for `Observable Properties`. By creating a systematic, evidence-based methodology, researchers can move the field from ad-hoc practices to a disciplined engineering approach. Though we have seen some early research about this topic (Shamim and Singhal, 2024), but there are a lot of work needed.

## 9 Threats to Validity

### 9.1 External Threats

Our initial framework identification process was dependent on the results of GitHub's keyword-based search API. This method is susceptible to search biases and may not capture all relevant repositories, posing a threat of an incomplete sample. To address this, we supplemented automated discovery with manual, domain-expert verification. For instance, the popular `langchain` framework was not returned by our initial

queries. However, our investigation revealed that the `langchain-ai` organization now advocates for `langgraph`[6] for building agentic systems, which was included in our final dataset. Similarly, `babyagi`, another popular framework in the research domain, was not included in our search results, and our manual analysis suggests that the framework's repository explicitly warns against production use.[7] These evidence suggest that while we cannot guarantee our list is exhaustive, this two-stage process of automated search followed by expert curation significantly reduces the risk of omitting key, production-relevant frameworks and strengthens the representativeness of our sample.

Our method for identifying agentic applications by searching for specific import statements presents two potential validity threats: (i) Any project that utilizes an agent framework through unconventional means, such as dynamic imports or custom wrappers that obscure the standard import statements, would not be included in our initial dataset. (ii) Noisy projects, e.g., projects that imported one or more agent frameworks for a minor, non-agentic utility, can be part of the dataset.

### 9.2 Construct Threats

To ground the architectural components of agent frameworks and agentic applications, we mapped each System Under Test (SUT) extracted from test functions to one of the canonical components defined in prior literature(Section 3.3). This mapping task introduces the possibility of rater bias. To mitigate this threat, we employed a closed card-sorting methodology with two independent raters assigning SUTs to predefined components. Disagreements were resolved through discussion, and inter-rater reliability was measured using Cohen's Kappa, yielding a substantial agreement score of $\kappa = 0.83$.

In comparing testing patterns across domains (e.g., software engineering, and ML applications), we faced inconsistencies in pattern granularity across source studies. Some testing patterns reported in the literature were more fine-grained than others (e.g., `equality` vs. `Assertion Based Testing`). To ensure a valid comparison, we normalized frequencies by converting raw counts to relative proportions within each domain, and we aggregated semantically similar sub-patterns into broader parent categories based on prior taxonomies (Openja et al., 2024). This allowed for consistent, meaningful comparison while preserving the underlying structure of each domain's testing practices.

## 10 Conclusion

This study provides the first large-scale empirical evidence of how practitioners are navigating the complex and uncertain terrain of testing FM-based AI agents. By analyzing a large-scale dataset of 39 agent frameworks and 439 agentic applications, we reveal that testing patterns in this domain reflect a pragmatic, yet incomplete, adaptation to the unique challenges posed by non-determinism and evolving architectural complexity.

[6] `https://github.com/langchain-ai/langchain?tab=readme-ov-file#langchains-ecosystem`

[7] `https://github.com/yoheinakajima/babyagi`

Our findings show that practitioners predominantly rely on adapted traditional testing strategies, e.g., assertion-based testing is complemented by negative testing and membership assertions in 40.2% of agent framework test functions and 36.7% of agentic application test functions. However, the limited adoption of domain-specific techniques (e.g., `DeepEval`, used in only 1.1% of agentic application test functions) indicates a notable gap in awareness and usability. Furthermore, we provide a list of 13 canonical components of agent architecture that can accommodate both existing and future components of agent frameworks and agentic applications. Mapping the testing patterns to these components, we reveal a critical blind spot, namely the near-total neglect of the `Trigger` component, which is covered in only around 1% of test functions. This component, serving as the prompt interface that directly governs the FM, exposes the entire ecosystem to risks of semantic drift, performance decay, and silent failures.

The significance of these findings is that the path to reliable AI agents does not lie in discarding established engineering principles, but in augmenting them with targeted, new methodologies. Our work lays the groundwork for more resilient agent development by offering a concrete empirical baseline and highlighting where testing effort must be rebalanced.

## Declaration

### Funding

Not applicable.

### Ethical Approval

This study does not involve human participants or animals.

### Informed Consent

Not applicable. No human subjects were involved in this study.

### Data Availability Statement

The dataset, experiment code, and experiment results of this study are available in our replication package.[8]

### Conflict of Interest

The authors declared that they have no known competing interests or personal relationships that could have (appeared to) influenced the work reported in this article.

[8] `https://github.com/SAILResearch/replication-25-agent-testing-empirical-study`

Clinical Trial Number in the Manuscript

Not applicable.

Author Contributions

- Mohammed Mehedi Hasan: Conceptualization, Data Collection, Methodology, Data Analysis, Writing – Original Draft.
- Hao Li: Methodology, Implication, Data Validation, Writing – Review & Editing.
- Emad Fallahzadeh: Writing – Review & Editing, Conceptual Guidance, Research Direction.
- Gopi Krishnan Rajbahadu: Data Validation, Writing – Review & Editing, Conceptual Guidance, Research Direction.
- Bram Adams: Writing – Review, Supervision, Research Direction.
- Ahmed E. Hassan: Supervision, Research Direction.

## References

X. Liu, H. Yu, H. Zhang, Y. Xu, X. Lei, H. Lai, Y. Gu, H. Ding, K. Men, K. Yang *et al.*, "Agentbench: Evaluating llms as agents," *arXiv preprint arXiv:2308.03688*, 2023.

I. Hettiarachchi, "Exploring generative ai agents: Architecture, applications, and challenges," *Journal of Artificial Intelligence General science (JAIGS) ISSN: 3006-4023*, vol. 8, no. 1, pp. 105–127, 2025.

J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, "Generative agents: Interactive simulacra of human behavior," in *Proceedings of the 36th annual acm symposium on user interface software and technology*, 2023, pp. 1–22.

Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu *et al.*, "Autogen: Enabling next-gen llm applications via multi-agent conversations," in *First Conference on Language Modeling*, 2024.

G. Mialon, C. Fourrier, T. Wolf, Y. LeCun, and T. Scialom, "Gaia: a benchmark for general ai assistants," in *The Twelfth International Conference on Learning Representations*, 2023.

S. Zhou, F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, T. Ou, Y. Bisk, D. Fried *et al.*, "Webarena: A realistic web environment for building autonomous agents," *arXiv preprint arXiv:2307.13854*, 2023.

X. Wang, Z. Wang, J. Liu, Y. Chen, L. Yuan, H. Peng, and H. Ji, "Mint: Evaluating llms in multi-turn interaction with tools and language feedback," *arXiv preprint arXiv:2309.10691*, 2023.

L. Weidinger, I. D. Raji, H. Wallach, M. Mitchell, A. Wang, O. Salaudeen, R. Bommasani, D. Ganguli, S. Koyejo, and W. Isaac, "Toward an evaluation science for generative ai systems," *arXiv preprint arXiv:2503.05336*, 2025.

R. Niedermayr, E. Juergens, and S. Wagner, "Will my tests tell me if i break this code?" in *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, 2016, pp. 23–29.

A. E. Hassan, D. Lin, G. K. Rajbahadur, K. Gallaba, F. R. Cogo, B. Chen, H. Zhang, K. Thangarajah, G. Oliva, J. Lin *et al.*, "Rethinking software engineering in the era of foundation models: A curated catalogue of challenges in the development of trustworthy fmware," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 294–305.

W. Ma, C. Yang, and C. Kästner, "(why) is my prompt getting worse? rethinking regression testing for evolving llm apis," in *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*, 2024, pp. 166–171.

S. Kokane, M. Zhu, T. M. Awalgaonkar, J. Zhang, A. Prabhakar, T. Q. Hoang, Z. Liu, R. RN, L. Yang, W. Yao *et al.*, "Toolscan: A benchmark for characterizing errors in tool-use llms," in *ICLR 2025 Workshop on Building Trust in Language Models and Applications*, 2025.

M. M. Hasan, H. Li, E. Fallahzadeh, G. K. Rajbahadur, B. Adams, and A. E. Hassan, "Model context protocol (mcp) at first glance: Studying the security and maintainability of mcp servers," 2025. [Online]. Available: https://arxiv.org/abs/2506.13538

A. Ehtesham, A. Singh, G. K. Gupta, and S. Kumar, "A survey of agent interoperability protocols: Model context protocol (mcp), agent communication protocol (acp), agent-to-agent protocol (a2a), and agent network protocol (anp)," *arXiv preprint arXiv:2505.02279*, 2025.

R. Lukyanenko, B. M. Samuel, J. Parsons, V. C. Storey, O. Pastor, and A. Jabbari, "Universal conceptual modeling: principles, benefits, and an agenda for conceptual modeling research," *Software and Systems Modeling*, vol. 23, no. 5, pp. 1077–1100, 2024.

T. Händler, "A taxonomy for autonomous llm-powered multi-agent architectures." in *KMIS*, 2023, pp. 85–98.

O. Boissier, R. H. Bordini, J. Hubner, and A. Ricci, *Multi-agent oriented programming: programming multi-agent systems using JaCaMo*. Mit Press, 2020.

H. Zhu, V. Terragni, L. Wei, S.-C. Cheung, J. Wu, and Y. Liu, "Understanding and characterizing mock assertions in unit tests," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 554–575, 2025.

L. Zamprogno, B. Hall, R. Holmes, and J. M. Atlee, "Dynamic human-in-the-loop assertion generation," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2337–2351, 2022.

M. Openja, F. Khomh, A. Foundjem, Z. M. Jiang, M. Abidi, and A. E. Hassan, "An empirical study of testing machine learning in the wild," *ACM Transactions on Software Engineering and Methodology*, 2024.

P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.

G. J. Myers, *The art of software testing*. John Wiley & Sons, 2006.

G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

B. Van Rompaey and S. Demeyer, "Exploring the composition of unit test suites," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*. IEEE, 2008, pp. 11–20.

Y. Tao, "An introduction to assertion-based verification," in *2009 IEEE 8th International Conference on ASIC*. IEEE, 2009, pp. 1318–1323.

C. Wei, L. Xiao, T. Yu, X. Chen, X. Wang, S. Wong, and A. Clune, "Automatically tagging the "aaa" pattern in unit test cases using machine learning models," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–3.

A. S. Rao, M. P. Georgeff *et al.*, "Bdi agents: from theory to practice." in *Icmas*, vol. 95, 1995, pp. 312–319.

G. Li, H. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, "Camel: Communicative agents for" mind" exploration of large language model society," *Advances in Neural Information Processing Systems*, vol. 36, pp. 51 991–52 008, 2023.

T. Händler, "Balancing autonomy and alignment: a multi-dimensional taxonomy for autonomous llm-powered multi-agent architectures," *arXiv preprint arXiv:2310.03659*, 2023.

Y. Liu, S. K. Lo, Q. Lu, L. Zhu, D. Zhao, X. Xu, S. Harrer, and J. Whittle, "Agent design pattern catalogue: A collection of architectural patterns for foundation model based agents," *Journal of Systems and Software*, vol. 220, p. 112278, 2025.

T. Masterman, S. Besen, M. Sawtell, and A. Chao, "The landscape of emerging ai agent architectures for reasoning, planning, and tool calling: A survey," *arXiv preprint arXiv:2404.11584*, 2024.

Y. Cheng, C. Zhang, Z. Zhang, X. Meng, S. Hong, W. Li, Z. Wang, Z. Wang, F. Yin, J. Zhao *et al.*, "Exploring large language model based intelligent agents: Definitions, methods, and prospects," *arXiv preprint arXiv:2401.03428*, 2024.

Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang, "Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

D. Gonzalez, J. C. Santos, A. Popovich, M. Mirakhorli, and M. Nagappan, "A large-scale study on the usage of testing patterns that address maintainability attributes: patterns for ease of modification, diagnoses, and comprehension," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 391–401.

C. Wei, L. Xiao, T. Yu, S. Wong, and A. Clune, "How do developers structure unit test cases? an empirical analysis of the aaa pattern in open source projects," *IEEE Transactions on Software Engineering*, 2025.

H. Zhu, L. Wei, V. Terragni, Y. Liu, S.-C. Cheung, J. Wu, Q. Sheng, B. Zhang, and L. Song, "Stubcoder: Automated generation and repair of stub code for mock objects," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–31, 2023.

A. Kampmann and A. Zeller, "Carving parameterized unit tests," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 248–249.

A. Fontes and G. Gay, "The integration of machine learning into automated test generation: A systematic mapping study," *Software Testing, Verification and Reliability*, vol. 33, no. 4, p. e1845, 2023.

J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024.

J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 1–36, 2020.

K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

Y. Nishi, S. Masuda, H. Ogawa, and K. Uetsuki, "A test architecture for machine learning product," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 273–278.

F. Tramer, V. Atlidakis, R. Geambasu, D. Hsu, J.-P. Hubaux, M. Humbert, A. Juels, and H. Lin, "Fairtest: Discovering unwarranted associations in data-driven applications," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 401–416.

M. Nejadgholi and J. Yang, "A study of oracle approximations in testing deep learning libraries," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 785–796.

R. Sahoo, S. Zhao, A. Chen, and S. Ermon, "Reliable decisions with threshold calibration," *Advances in Neural Information Processing Systems*, vol. 34, pp. 1831–1844, 2021.

T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," *arXiv preprint arXiv:2402.01680*, 2024.

L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin *et al.*, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024.

Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation framework," *arXiv preprint arXiv:2308.08155*, 2023.

S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou *et al.*, "Metagpt: Meta programming for multi-agent collaborative framework," *arXiv preprint arXiv:2308.00352*, vol. 3, no. 4, p. 6, 2023.

D. Gonzalez, T. Zimmermann, and N. Nagappan, "The state of the ml-universe: 10 years of artificial intelligence & machine learning software development on github," in *Proceedings of the 17th International conference on mining software repositories*, 2020, pp. 431–442.

H. Li and C.-P. Bezemer, "Bridging the language gap: an empirical study of bindings for open source machine learning libraries across software package ecosystems," *Empirical Software Engineering*, vol. 30, no. 1, p. 6, 2025.

E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 92–101.

N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, pp. 3219–3253, 2017.

J. Han, S. Deng, X. Xia, D. Wang, and J. Yin, "Characterization and prediction of popular projects on github," in *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, vol. 1. IEEE, 2019, pp. 21–26.

B. Okken, *Python Testing with pytest*. Pragmatic Bookshelf, 2022.

A. Bodea, "Pytest-smell: a smell detection tool for python unit tests," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 793–796.

B. Cui, J. Li, T. Guo, J. Wang, and D. Ma, "Code comparison system based on abstract syntax tree," in *2010 3rd IEEE International Conference on Broadband*

*Network and Multimedia Technology (IC-BNMT)*. IEEE, 2010, pp. 668–673.
E. Spirin, E. Bogomolov, V. Kovalenko, and T. Bryksin, "Psiminer: A tool for mining rich abstract syntax trees from code," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 13–17.
Z. Zhao, Y. Chen, A. A. Bangash, B. Adams, and A. E. Hassan, "An empirical study of challenges in machine learning asset management," *Empirical Software Engineering*, vol. 29, no. 4, p. 98, 2024.
M. M. Hassan and A. Rahman, "As code testing: Characterizing test quality in open source ansible development," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 208–219.
S. Gueron, S. Johnson, and J. Walker, "Sha-512/256," in *2011 Eighth International Conference on Information Technology: New Generations*. IEEE, 2011, pp. 354–358.
D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.
F. Zampetti, R. Kapur, M. Di Penta, and S. Panichella, "An empirical characterization of software bugs in open-source cyber–physical systems," *Journal of Systems and Software*, vol. 192, p. 111425, 2022.
G. Meszaros, S. M. Smith, and J. Andrea, "The test automation manifesto," in *Conference on extreme programming and agile methods*. Springer, 2003, pp. 73–81.
Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 214–224.
M. J. Parker, C. Anderson, C. Stone, and Y. Oh, "A large language model approach to educational survey feedback analysis," *International journal of artificial intelligence in education*, pp. 1–38, 2024.
R. Patil, S. Boit, V. Gudivada, and J. Nandigam, "A survey of text representation and embedding techniques in nlp," *IEEE Access*, vol. 11, pp. 36 120–36 146, 2023.
J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.
A. Bhargava, C. Witkowski, A. Detkov, and M. Thomson, "Prompt baking," *arXiv preprint arXiv:2409.13697*, 2024.
S. Fujita, Y. Kashiwa, B. Lin, and H. Iida, "An empirical study on the use of snapshot testing," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023, pp. 335–340.
W. Lam, S. Srisakaokul, B. Bassett, P. Mahdian, T. Xie, P. Lakshman, and J. De Halleux, "A characteristic study of parameterized unit tests in. net open source projects," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018, pp. 5–1.
Z. Wan, X. Xia, D. Lo, and G. C. Murphy, "How does machine learning change software development practices?" *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1857–1871, 2019.
Y. Liu, D. Iter, Y. Xu, S. Wang, R. Xu, and C. Zhu, "G-eval: Nlg evaluation using gpt-4 with better human alignment," *arXiv preprint arXiv:2303.16634*, 2023.
S. Es, J. James, L. E. Anke, and S. Schockaert, "Ragas: Automated evaluation of retrieval augmented generation," in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, 2024, pp. 150–158.

L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46 595–46 623, 2023.

J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

A. Confident, "Deepeval," 2024.

F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

N. Tillmann and W. Schulte, "Parameterized unit tests," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 253–262, 2005.

Z. Zhang, X. Bo, C. Ma, R. Li, X. Chen, Q. Dai, J. Zhu, Z. Dong, and J.-R. Wen, "A survey on the memory mechanism of large language model based agents," *arXiv preprint arXiv:2404.13501*, 2024.

F. Bang, "Gptcache: An open-source semantic cache for llm applications enabling faster answers and cost savings," in *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, 2023, pp. 212–218.

Q. Zhang, M. Wornow, and K. Olukotun, "Cost-efficient serving of llm agents via test-time plan caching," *arXiv preprint arXiv:2506.14852*, 2025.

G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, "Prompt engineering in large language models," in *International conference on data intelligence and cognitive informatics*. Springer, 2023, pp. 387–402.

R. C. Barron, V. Grantcharov, S. Wanna, M. E. Eren, M. Bhattarai, N. Solovyev, G. Tompkins, C. Nicholas, K. Ø. Rasmussen, C. Matuszek *et al.*, "Domain-specific retrieval-augmented generation using vector stores, knowledge graphs, and tensor factorization," in *2024 International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2024, pp. 1669–1676.

N. Nascimento, P. Alencar, and D. Cowan, "Self-adaptive large language model (llm)-based multiagent systems," in *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 2023, pp. 104–109.

N. Rajaraman, J. Jiao, and K. Ramchandran, "Toward a theory of tokenization in llms," *arXiv preprint arXiv:2404.08335*, 2024.

X. Li, S. Wang, S. Zeng, Y. Wu, and Y. Yang, "A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges," *Vicinagearth*, vol. 1, no. 1, p. 9, 2024.

Z. Wu, X. Lin, Z. Dai, W. Hu, Y. Shu, S.-K. Ng, P. Jaillet, and B. K. H. Low, "Prompt optimization with ease? efficient ordering-aware automated selection of exemplars," *Advances in Neural Information Processing Systems*, vol. 37, pp. 122 706–122 740, 2024.

A. Ajibode, A. A. Bangash, F. R. Cogo, B. Adams, and A. E. Hassan, "Towards semantic versioning of open pre-trained language model releases on hugging face," *Empirical Software Engineering*, vol. 30, no. 3, pp. 1–63, 2025.

I. Shamim and R. Singhal, "Methodology for quality assurance testing of llm-based multi-agent systems," in *Proceedings of the 4th International Conference on AI-ML Systems*, 2024, pp. 1–5.

## A Example of Testing Patterns

### A.1 Structural Patterns

#### *A.1.1 Hyperparameter Control*

Listing 1: Hyperparameter Control: On line 6 hyperparameter temperature is set at 0.

```
1 def test_10_concurrent_API_calls(self):
2     tools = []
3     with open('./data/schemas/get-headers-params.json', 'r') as f:
4         tools = ToolFactory.from_openapi_schema(f.read(), {})
5     ceo = Agent(name='CEO', tools=tools, instructions="You are an agent that tests
          concurrent API calls. You must say 'success' if the output contains
          headers, and 'error' if it does not and **nothing else**.")
6     agency = Agency([ceo], temperature=0)
7     result = agency.get_completion("Please call PrintHeaders tool TWICE at the
          same time in a single message. If any of the function outputs do not
          contains headers, please say 'error'.")
8     self.assertTrue(result.lower().count('error') == 0, agency.main_thread.
          thread_url)
```

#### *A.1.2 Parameterized Testing*

Listing 2: Parameterized Testing: different llm models have been passed to the same test function through parameter.

```
1 @pytest.mark.parametrize('llm', ['transformers:gpt2', 'transformers:facebook/opt-350m'])
2 def test_custom_kwargs_transformers(llm):
3     """Test if we can pass model specific kwargs."""
4     llm = get_llm(llm)
5     program = engine("Repeat the following 10 times: Repeat this. Repeat this.
          Repeat this. Repeat this.{{gen 'completion' max_tokens=4
          repetition_penalty=10.0}}", llm=llm)
6     executed_program = program()
7     assert not executed_program['completion'].startswith(' Repeat this.')
```

#### *A.1.3 Test Double*

Listing 3: A testdouble or Mock is setup to replicate the behavior of an MCP server

```
1 @pytest.mark.asyncio
2 async def test_mcp_error_handling(self):
3     """Test MCP connection error handling."""
4     with patch('mcp.client.stdio.stdio_client') as mock_stdio_client:
5         mock_stdio_client.side_effect = ConnectionError('Failed to connect to MCP
              server')
6         try:
7             async with mock_stdio_client(Mock()) as (read, write):
8                 pass
9             assert False, 'Should have raised ConnectionError'
```

```
        except ConnectionError as e:
            assert 'Failed to connect to MCP server' in str(e)
```

## A.2 Verification Patterns

### *A.2.1 Assertion Based Testing*

Listing 4: Assertion Based Testing where equality is being verified.

```
def test_init(self, memory):
    assert memory.name == 'MyMemory'
```

### *A.2.2 DeepEval*

Listing 5: DeepEval: Using DeepEval for verifying the company name name in a job description by a resume retriever agent.

```
def test_relevant_content_retrieved(user_proxy: UserProxyAgent,
     resume_retriever_agent: ResumeRetriever, job_description: str, company: str)
      -> None:
    message = "Here is the job description: " + job_description
    chat_outcome = get_chat_outcome(user_proxy, resume_retriever_agent, message)
    assert_test(
        LLMTestCase(
            input=message,
            actual_output=chat_outcome,
            context=[company]
        ),
        [
            GEval(
                name="Inclusion",
                evaluation_params=[
                    LLMTestCaseParams.INPUT,
                    LLMTestCaseParams.ACTUAL_OUTPUT,
                ],
                threshold=0.7,
                evaluation_steps=[
                    f"Check that the output contains the company name '{company}'",
                    f"Check that the output does not contain any company name other
                          than '{company}'",
                ],
            )
        ],
    )
```

### *A.2.3 Membership Testing*

Listing 6: Membership Testing: verifying whether a string or pattern is member of the result.

```
def test_task_guardrail_process_output(task_output):
```

```
    guardrail = LLMGuardrail(description='Ensure the result has less than 10 words
        ', llm=LLM(model='gpt-4o'))
    result = guardrail(task_output)
    assert result[0] is False
    assert 'exceeding the guardrail limit of fewer than' in result[1].lower()
    guardrail = LLMGuardrail(description='Ensure the result has less than 500
        words', llm=LLM(model='gpt-4o'))
    result = guardrail(task_output)
    assert result[0] is True
    assert result[1] == task_output.raw
```

### *A.2.4 Mock Assertion*

Listing 7: Mock Assertion: verifying whether the a method from the mocked object is invoked while testing the process_request method.

```
async def test_pii_detection_blocking(self):
    """Test that content with PII is blocked"""
    self.mock_comprehend_client.detect_pii_entities.return_value = {'Entities': [{
        'Type': 'EMAIL', 'Score': 0.99}, {'Type': 'PHONE', 'Score': 0.95}]}
    response = await self.agent.process_request(input_text='Contact me at
        test@email.com', user_id='test_user', session_id='test_session',
        chat_history=[])
    self.assertIsNone(response)
    self.mock_comprehend_client.detect_pii_entities.assert_called_once()
```

### *A.2.5 Negative Test*

Listing 8: Negative Test: test for handling wrong start node in Mermaid code

```
def test_mermaid_code_start_wrong():
    with pytest.raises(LookupError):
        graph1.mermaid_code(start_node=Spam)
```

### *A.2.6 Snapshot Testing*

Listing 9: Snapshot Assertion: markdown slash-command testing by comparing with snapshots.

```
def test_handle_slash_command_markdown():
    io = StringIO()
   assert handle\_slash\_command('/markdown', [], False, Console(file=io), '
       default') == (None, False)
    assert io.getvalue() == snapshot('No markdown output available.\n')
    messages: list[ModelMessage] = [ModelResponse(parts=[TextPart('[hello](\#
        hello)'), ToolCallPart('foo', '\{\}')])]
    io = StringIO()
    assert handle_slash_command('/markdown', messages, True, Console(file=io), '
        default') == (None, True)
    assert io.getvalue() == snapshot('Markdown output of last question:\n\n[hello](# hello)\n')
```

### *A.2.7 Value Range Analysis*

Listing 10: Value Range Analysis: verifying whether the output is within a predefined range

```
def test_path_traversal_characters():
    filename = '../../etc/passwd'
    sanitized = sanitize_filename(filename)
    assert sanitized.startswith('passwd_')
    assert len(sanitized) <= MAX_FILENAME_LENGTH
```