Future of AI is Multi-Agent with Sub-Agents in Software Testing and Automation

Him Raj Singh*

* Staff Software Engineer, PayPal Inc.

DOI: 10.29322/IJSRP.16.02.2026.p17046
https://dx.doi.org/10.29322/IJSRP.16.02.2026.p17046

Paper Received Date: 17th January 2026
Paper Acceptance Date: 18th February 2026
Paper Publication Date: 24th February 2026

Abstract- Modern AI is shifting from a single "all-knowing" agent to networks of collaborating agents. In software testing and automation, this means deploying specialized sub-agents – each focused on tasks like UI testing, API validation, or security checks – under a coordinating orchestrator. This article explores core concepts of multi-agent AI, the roles of sub-agents in testing workflows, and the benefits they bring (better coverage, parallel execution, higher quality). We also examine the limitations and risks (complexity, cost, coordination overhead) and challenges of monolithic single-agent approaches (expertise dilution, bottlenecks). Industry examples like Anthropic's Claude Code subagents and GitHub Copilot illustrate current multi-agent trends. Finally, we discuss future directions (standardized protocols, advanced orchestrators, agent ecosystems) for AI-driven testing. Throughout, we cite recent research and data to support each point.

Index Terms- Multi-Agent Systems, AI in Software Testing, Sub-Agents, Model Context Protocol (MCP), AI-Assisted Development

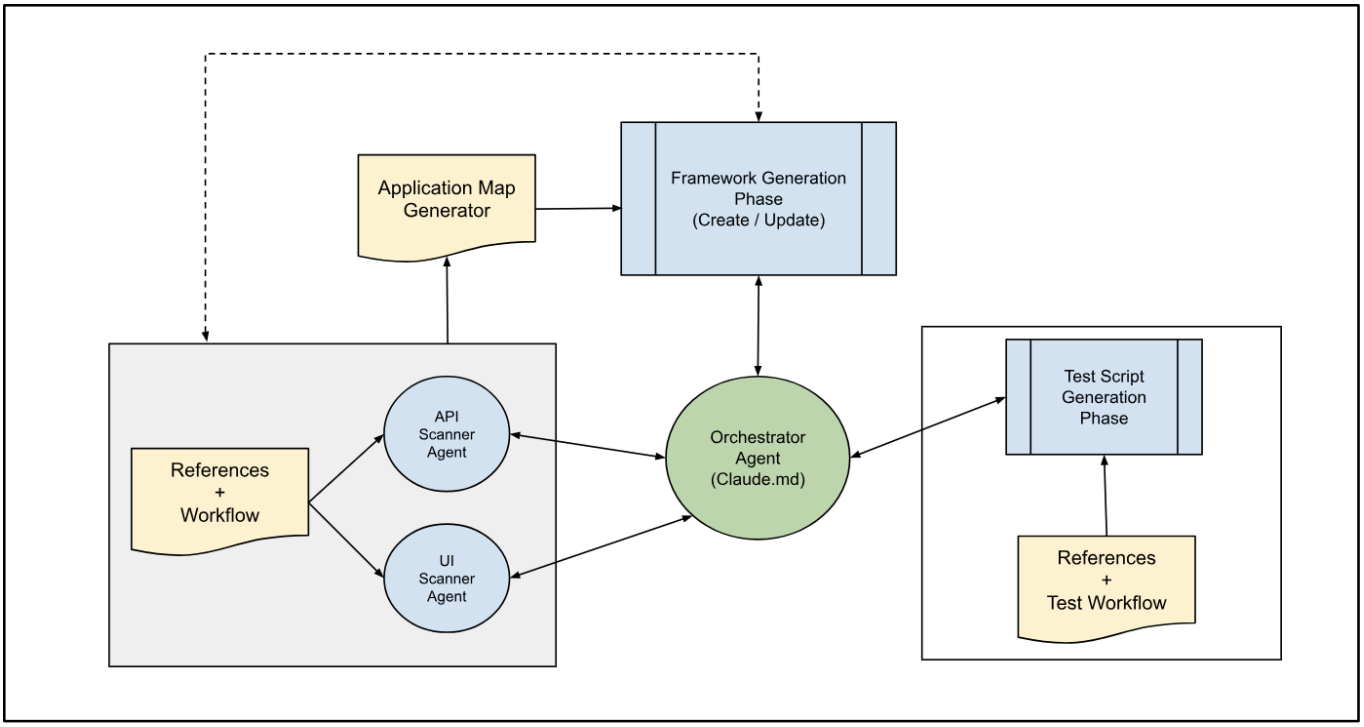
I. INTRODUCTION

Multi-agent AI refers to systems where multiple intelligent agents coordinate to solve complex tasks. Rather than one model handling everything, a lead or orchestrator delegates subtasks to specialized sub-agents. Each sub-agent has its own context, tools, and "expertise" tuned for a specific domain. This mirrors how human teams work: each member focuses on a specialty while a manager integrates their work. By contrast, single-agent models try to be generalists, which often fails at scale.

Leading tech firms recognize this shift. For example, Microsoft notes that enterprise AI is moving from "single 'do-everything' agents" to *collections of autonomous, task-specialized agents* coordinated by an orchestrator. Each agent can combine a large or small LLM core with domain-specific tools and memory. The orchestrator preserves overall context and routes tasks to the right agents. Anthropic's research team similarly found that *multi-agent systems* can outperform a single agent on complex research tasks by parallelizing work in separate context windows. In their tests, a Claude Opus 4 "lead agent" that spawned Claude Sonnet 4 subagents achieved 90.2% better performance on open-ended queries than the single-agent baseline. In software testing, where workflows are complex and stateful, this multi-agent approach promises to handle breadth and depth that single agents miss.

Claude Code (Anthropic's coding assistant) and other platforms now offer **subagents** explicitly. A subagent is a pre-configured AI instance with a narrow role: its own prompt, its own toolset, and its own context window. For example, Claude Code lets you create a "TestAgent" subagent that knows how to run Pytest, or a "CodeReviewer" subagent trained to catch style issues. Each subagent operates independently, so it can specialize without cluttering the main agent's context. This modularity is key: by assigning **one agent one job**, we ensure focused expertise and clear separation of concerns. As one guide puts it, multi-agent systems allow "context management" for long tasks, parallel workers for sub-tasks, and distributed development by different teams

As part of my research, I used Claude Code subagent to build Web/API Test Automation sub-agent to perform Automation framework creation/updation plus Automating the test scenarios provided as input to the agent via .md files. This agent project was created for a generic use by any Automation testers and was pushed to Git for any member in the team to utilize. I also created the marketplace plugin for this agent to be readily available for anyone to use. Below is the architectural flow diagram for the sub-agent that I built.



II. Sub-Agent Roles And Workflows In Testing

In software testing, multi-agent workflows assign each testing domain to a sub-agent. For instance, a complex web application might be tested by:

- **UI Testing Agent:** Focuses on front-end validation (cross-browser, responsive UI, user flows).
- **API Testing Agent:** Validates REST/GraphQL endpoints, service integration, and backend logic.
- **Database Testing Agent:** Checks data integrity, performs query performance tests, and assesses schema changes.
- **Security Testing Agent:** Looks for vulnerabilities (injection, auth flaws, encryption), enforcing compliance standards.
- **Performance Testing Agent:** Simulates load and measures response times across components.
- **Integration Coordination Agent:** Orchestrates end-to-end scenarios, managing dependencies between subsystems.

Each of these specialized agents "focuses on specific testing domains" while communicating results to others. For example, Virtuoso QA describes UI testing agents with "cross-browser intelligence" and self-healing of UI elements, and API agents that validate endpoints and data flow. In practice, a testing pipeline might work as follows: the orchestrator (or lead agent) receives a new code deployment to test; it then **spawns subagents** for UI, API, etc., and passes relevant context to each (e.g. UI selectors to the UI agent, endpoint definitions to the API agent). These agents run their tests in parallel – for example, the UI agent executes Playwright scripts across browsers, while the API agent runs Postman or pytest-based integration tests. Each subagent returns its findings (pass/fail, bugs discovered) to the orchestrator. A final coordinating agent can then synthesize these results into a comprehensive report and decide if further testing is needed.

This workflow is akin to Claude Code's subagents in development. As a guide notes, teams can create subagents like *"The Code Reviewer"* or *"The Test Engineer,"* each armed with their own tools (e.g. ESLint, Jest). In testing, one could analogously build agents such as a *"Smoke Test Agent"* with Selenium, or a *"DB Validator"* with SQL clients. The main orchestrator routes tasks: if code changes include UI updates, it invokes the UI agent; if there are new APIs, it routes to the API agent. All agents share results via a common

interface so that, for example, the Integration Coordination Agent can verify that UI actions correctly trigger API calls and database updates.

By dividing the workload, multi-agent testing pipelines avoid overloading any single model's context. Each sub-agent only sees relevant information for its domain, which improves focus and accuracy. Moreover, agents can be invoked explicitly or automatically when their expertise is needed. For example, a developer might explicitly command, "Use the security-agent to scan for SQL injection," or the orchestrator may automatically select it when API endpoints change. Advanced implementations might even support *living documentation*: an agent continuously updates API specs or test documentation as code evolves.

III. Benefits of Multi-Agent Testing Systems

Deploying multiple specialized AI agents yields significant advantages over monolithic approaches. First, **task specialization** ensures expert-level validation in each domain. As one guide explains, assigning "clear role to each sub-agent" means a security-focused agent will catch issues a generalist might miss. In practice, this reduces blind spots. Virtuoso QA claims that using multi-agent testers makes application validation "94% more effective" by covering more components simultaneously. Industry surveys indicate that *79% of production incidents involve cross-system interactions* – scenarios where single-domain testers fail. Specialized agents can collaboratively cover these cross-cutting cases, catching integration defects early.

Second, multi-agent setups enable **parallel processing and scalability**. Independent agents can run tests concurrently: a UI test runs on one agent while an API test runs on another. This can drastically **speed up testing cycles**. A sub-agent-based workflow "can slash project timelines" by parallelizing linting, testing, and documentation tasks. As projects grow, new subagents can be spun up for new features or platforms, without overloading any one agent. This *infinite scalability* is akin to adding more workers to a factory assembly line. Empirical benchmarks support this: one LangChain study found that adding multi-agent orchestration improved performance by ~50% on complex tasks compared to a single agent with the same inputs. In fact, with many distractor domains, the **single-agent performance fell off sharply** while a "swarm" of agents maintained high accuracy.

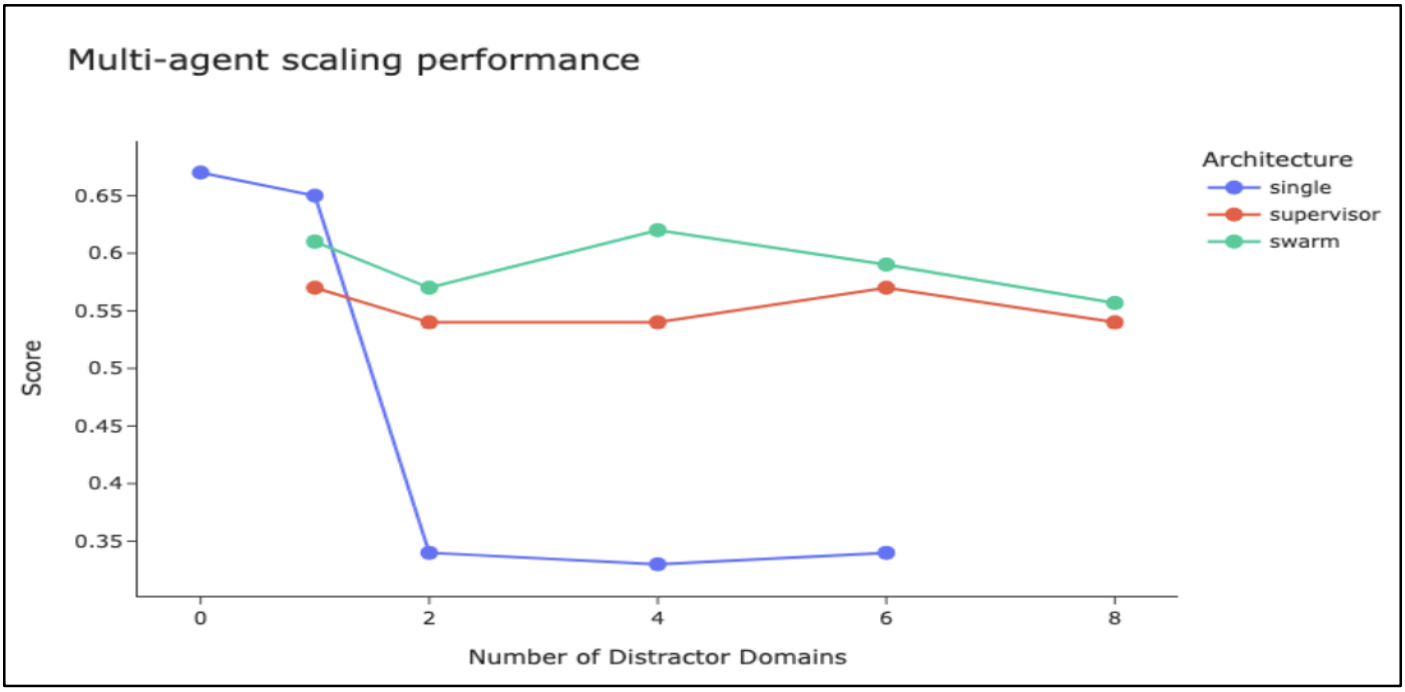


Figure: Performance vs. number of distractor domains for different agent architectures. A single-agent (blue) quickly degrades as task complexity grows, whereas multi-agent setups (swarm in green, supervisor in red) maintain higher accuracy.

Third, multi-agent testing enhances **coverage and depth** of analysis. Multiple agents can explore different facets of the system in breadth-first manner. Anthropic notes that subagents "operate in parallel with their own context windows, exploring different aspects of the question simultaneously". In testing, this means one agent might generate regression test cases while another focuses on edge-case fuzzing at the same time. The orchestrator can then merge their findings for a more complete picture. Additionally, multi-agent systems can handle larger "context" overall: each agent has a separate window, effectively expanding total memory. Anthropic reported that

multi-agent architectures "scale token usage for tasks that exceed single-agent limits", implying they can handle more complex test specifications.

Fourth, multi-agent approaches foster **collaboration and consistency**. Teams can share and reuse subagent configurations (e.g. a "Frontend Linter" agent used across projects). This standardizes quality and spreads best practices. For example, one project's Security-Agent can be adopted in another team, ensuring consistent coverage. In practice, some platforms allow versioning and sharing of agent profiles.

Finally, there are tangible economic and reliability benefits. As Virtuoso QA points out, *enterprises lose millions to integration failures that single-agent testing misses*. By catching these issues early with a multi-agent strategy, organizations can avoid costly production incidents. Anthropic's data shows that beyond performance, groups of agents can "accomplish far more" than lone agents as tasks grow in complexity. By exploiting parallelism and specialist knowledge, multi-agent systems deliver higher-quality results (and can justify their added compute cost by reducing expensive downtime).

IV. Limitations and Risks of Multi-Agent Systems

Despite their promise, multi-agent testing systems have downsides that must be managed. The most obvious is **computational cost**. Anthropic notes that multi-agent chains use *far* more tokens than single-chat interactions – on the order of 4× more per task, and roughly 15× more in their experiments. Running many agents in parallel can thus increase cloud/API costs significantly. In LangChain's benchmarks, interestingly, multi-agent (swarm/supervisor) used **fewer tokens** than a single agent with distractors, but this depends on task structure. In general, spawning extra agents means more context windows to fill, which can become expensive. Teams must ensure the performance gains justify the token/resource usage – for low-value tasks, the cost overhead can outweigh benefits.

Multi-agent systems also introduce **architectural complexity**. Coordinating many agents is fundamentally harder than a single pipeline. Agents can **step on each other's toes** or duplicate work if not well orchestrated. For example, ensuring consistent shared state (like test data or user sessions) across agents is non-trivial. Virtuoso QA highlights challenges like "parallel validation requirements" and "cross-system test dependencies" that monolithic agents can't handle easily – but even in multi-agent setups, the orchestrator must carefully sequence tests (e.g. ensuring the UI agent tests only after the API agent has set up data). Complex handoff protocols must preserve context and error information.

There are also **emergent risk factors**. Each subagent is a machine learning model that may hallucinate or make unexpected choices. When agents interact, unforeseen behaviors can emerge. As Mabl points out, traditional testing assumes predictable flows, but multi-agent interactions are dynamic and unpredictable. For instance, if one agent fails mid-test (e.g. a service agent times out), the system needs graceful recovery strategies. Likewise, designing consistent error handling across agents is hard. Without proper monitoring, errors can be obscured in agent chatter.

Security and governance are further concerns. Multiple AI components means a larger attack surface: each agent has access to different data and tools. Ensuring least-privilege access (e.g. only the DB agent can query the database) is crucial. Sharing secrets or API tokens across agents must be done securely. Additionally, debugging multi-agent failures can be difficult, since logs and decisions are distributed. Standardizing logging and observability (for example, using shared protocols like Model Context Protocol, MCP) becomes essential to trace what each agent did.

Finally, integrating multi-agent testing into existing pipelines poses practical challenges. Teams need to set up orchestration infrastructure, maintain many agent models and prompts, and manage inter-agent communication. There is a risk of fragmentation: if each agent evolves separately, the overall system can become incoherent. Over-specialization is another potential pitfall: if agents are too narrow, some edge-case tests might fall through the cracks. Thus, rigorous design, monitoring, and possibly human-in-the-loop oversight are required to mitigate these risks.

V. Challenges of Single-Agent Approaches

The move toward multi-agent is largely driven by the shortcomings of single-agent systems in testing complex software. Monolithic AI testers suffer from **expertise dilution**. As Virtuoso QA notes, a lone AI trying to handle UI, API, security, and performance ends up shallow on each: "domain expertise dilution" causes gaps in coverage. The agent must constantly context-switch between test types, losing focus. In practice, this means single agents miss integration bugs: one estimate is that *79% of critical incidents involve multi-system interactions that monolithic testing misses*.

Single-agent systems also **do not scale well**. One agent has a finite context window and throughput; as application complexity grows, it becomes a bottleneck. Microsoft warns that a single agent architecture "collapses under the weight of real-world enterprise constraints," leading to slower performance as more users and tasks accumulate. LangChain's benchmarks confirm this: when irrelevant context or many tools are available, single-agent performance rapidly deteriorates. In their study, adding just one extra domain caused a single agent's accuracy to plummet, whereas a multi-agent swarm held steady.

Coordination gaps are another flaw. A single agent cannot easily manage complex workflows that span multiple layers. Virtuoso QA highlights issues like "cross-system test dependencies" and "parallel validation requirements" that monolithic agents struggle with. For example, if a UI change must trigger backend API tests and database validation, a single agent would have to serially perform all steps in one flow, risking missed timing or inconsistent state. With one agent, you also can't easily reuse results or parallelize tasks.

Finally, there are **governance and evolution** problems. In a single-agent system, updating or adding a new capability often means retraining or rewriting one huge prompt. This "change management complexity" slows innovation. Security is also harder: a single agent typically has blanket access to all data and tools, violating the principle of least privilege. Overall, enterprises report that rigid, centralized AI testers with one agent lead to *slower innovation, higher operational risk, and diverging from best practices*. These challenges make the case clear: complex testing calls for modular, distributed agent teams, not one-size-fits-all solutions.

VI. Future Directions

The trend toward multi-agent testing will accelerate as tooling and standards improve. We expect **more sophisticated orchestration layers** that handle context and communication seamlessly. Frameworks like LangChain, Microsoft Semantic Kernel, and Anthropic's interleaved-thinking are evolving to better coordinate subagents. Standard protocols (like MCP) will likely emerge so agents from different vendors can interoperate.

In software testing specifically, we'll see richer **test agent ecosystems**. Platforms may offer libraries of pre-built subagents (UI tester, security scanner, etc.) that teams can configure. These subagents might integrate with CI/CD pipelines: for instance, a commit could trigger a workflow that auto-spawns relevant testing agents, runs them in parallel in the cloud, and aggregates results. We may also see agents with **learned expertise** – using reinforcement learning or retrospective analysis to improve over time. For example, if a flake fails repeatedly in CI, a "flaky-test agent" could analyze and suggest fixes or workarounds.

AI itself will aid in building these systems. Tools like ChatDev and AutoGen (in IBM's and Microsoft's stacks) generate multi-agent plans from high-level goals. Future agents might be capable of *creating* new subagents on the fly for novel scenarios – for example, spinning up a "mobile testing agent" if a new platform is detected. Memory and long-term knowledge sharing will grow; agents could maintain shared logs or learned models to remember past bugs and avoid regressions.

Industry momentum is strong. Anthropic states that subagents "are a glimpse into the future of software development, where we act as conductors for a team of specialized AIs". Microsoft calls the shift away from single agents a "strategic imperative" for scalable AI. We anticipate developments such as standardized agent-communication languages, federated agent networks (for distributed teams), and regulated compliance checks integrated into agents. As models improve and costs decline, generic multi-agent architectures will likely become the norm.

In summary, the future of AI-driven testing is **collective intelligence**. By harnessing sub-agents, software teams can build intelligent testing ecosystems that match the complexity of modern applications. This emerging paradigm promises faster, more thorough QA, but requires careful engineering to manage the new complexity. As multi-agent tools mature, they will unlock a new era of automated testing – one in which human testers become overseers of AI teams, rather than sole executors of every test.

VII. Conclusion

AI in software testing is moving beyond solo agents to dynamic **multi-agent systems**. By assigning specialized sub-agents to UI tests, API checks, security scanning, etc., teams achieve deeper coverage and speed through parallelism. Industry data and benchmarks confirm substantial gains: multi-agent setups dramatically improve fault detection and resilience compared to monolithic AI testers. However, this comes with trade-offs in cost and complexity. Robust orchestration, observability, and governance are crucial to reap the benefits while avoiding new risks.

The journey from one "super-agent" to an AI *team* is already underway. Tools like Claude Code and Copilot exemplify modular AI assistants, and enterprise platforms are building out full agent frameworks. As Anthropic notes, we are becoming "conductors for a team of specialized AIs". In the near future, standardized protocols and more mature agent ecosystems will make these systems even more

powerful and reliable. Ultimately, embracing multi-agent AI in testing promises a leap in software quality – provided we design carefully for the challenges. The future is clearly a *team* of AIs collaborating to build and validate software, with human engineers guiding the orchestra.

REFERENCES

[1] https://code.claude.com/docs/en/sub-agents
[2] Li, X., Wang, S., Zeng, S., Wu, Y., Yang, Y., *et al.* "A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges." *Artificial Intelligence Review*, vol. 1, no. 9, 2024. A systematic survey of LLM-based agent systems, key components, architectures, and open challenges in multi-agent workflows.
[3] Naqvi, S., Baqar, M., Nawaz, A. M. "The Rise of Agentic Testing: Multi-Agent Systems for Robust Software Quality Assurance," *arXiv preprint*, Jan 5, 2026. Introduces an agentic multi-model testing framework with collaborative agents for continuous testing and self-correcting loops.
[4] Tian, F., Luo, A., Du, J., *et al.* "An Outlook on the Opportunities and Challenges of Multi-Agent AI Systems," *arXiv preprint*, May 23, 2025. Provides a foundational framework for analyzing multi-agent systems' effectiveness and safety.
[5] Calegari, R., Ciatto, G., Mascardi, V., *et al.* "Logic-based technologies for multi-agent systems: a systematic literature review," *Autonomous Agents and Multi-Agent Systems*, vol. 35, no. 1, Springer, 2020. A foundational review of multi-agent system technologies and cooperation/coordination mechanisms.
[6] *Multi Agent Systems: Studying Coordination and Cooperation Mechanisms in Multi-Agent Systems to Achieve Collective Goals Efficiently, Journal of Artificial Intelligence Research*, vol. 4, no. 1, Feb 2024. Discusses coordination and cooperation principles in multi-agent systems.
[7] https://www.virtuosoqa.com/post/multi-agent-testing-systems-cooperative-ai-validate-complex-applications#:~:text=Cross,agent%20capabilities

AUTHORS

First Author – Him Raj Singh, Bachelor of Technology in Computer Science and Engineering, himrajaccess@gmail.com