International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group| AECTIC2025-Conference Proceeding
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.AECTIC-118

Original Article

Agentic AI for Software Development: Autonomous Agents in Requirements Engineering, Testing, and Deployment

Ravikanth Konda
Software Application Engineer.

Abstract - *The maturation of agentic AI (autonomous large language model (LLM) agents with planning, tool-use, and memory) is a sea change in software engineering. In contrast to the earlier, AI-guided tools that just presented static suggestions or local completions, AGI AI possesses the capability of goal-directed multi-step reasoning across the entire SDLC. This is most obviously seen in three of the mission-critical applications: requirements engineering, testing, and deployment. Each of these stages has long been identified as an area of bottleneck or waste for software quality and delivery speed, and can be subject to inefficiency that autonomous agents can methodically diminish under suitable constraints and evaluation. Such an agentic AI can parse user stories, epics, and regulatory docs in RE, surfacing ambiguities, contradictions, and compliance risks proactively. By simulating a requester Analyst's position, such agents support the communication with stakeholders and suggest some useful clarification questions that are automatically proposed, with an eye on compliant rules to standards (e.g., SSTC 29148). This feature speeds up not only backlog refinement, but also traceability between requirements and test artifacts, as well as implementation units. In software testing, the benefits of independence are even clearer. Classical automatic testing techniques use fixed heuristics, or user-guided scenarios, whereas autonomic test generators can generate heterogeneous sets of unit, integration, and property-based test cases by iterating over the refinement of coverage goals. By combining fuzzing engines, mutation coverage, and continuous test adequacy feedback, agents can dynamically exercise system interfaces to visit boundary conditions and patch weak assertions that, in turn, lower defect leakage rates.*

This is consistent with advances in an emerging trend---repository-level evaluation (e.g., SWE-bench, DevBench), where iterative reasoning and environment-aware feedback loops are again the key to success across settings against static baseline models. Lastly, in deployment and operations, agentic AI operates as a release manager and reliability engineer. And such autonomous agents or systems should help out when we roll – deploying new versions, watching the real-time observability data, and kicking off roll-back procedures when they see problems – in consideration of a progressive delivery approach (ie, canary or blue-green). Contrary to rule-based automation, agentic systems adjust rollout decisions based on changing telemetry patterns and serve policy-as-code checks for safety. They can also independently propose and document planned mitigation actions, incident traces, and coordinate advanced rollback windows that help to minimize MTTD (Mean Time To Detection) and MTTR across production failures. In this paper, we contribute a role-based multi-agent architecture designed for these three lifecycle pillars, as well as a method for robust integration using retrieval grounding, structured outputs, and reflexive critique, and we evaluate empirically over benchmark datasets and industrial microservices. Results show that there is a measurable improvement in ambiguity reduction in requirements of 42%, mutation score increase of 14% in testing, and rollback latency decrease of 37% with deployment without the requirement to accept an elevated change failure rate. Beyond the empirical gains, we have also shown socio-technical implications illustrating that there is a need for governance to support human-in-the-loop checks as well as open audit systems. Taken together, the results imply that agentic AI is not just an accelerant but a structural enabler that offers a dependable basis to reconcile autonomy and safety in contemporary software development pipelines.

Keywords - *Agentic AI, Requirements Engineering, Software Testing, Deployment Automation, Large Language Models, Multi-Agent Systems, DevOps; GitOps, Canary Releases, Mutation Testing, Traceability, Reflexive Critique, Policy-as-Code, Autonomous Software Agents, Progressive Delivery.*

1. Introduction

Large language models (LLMs) have introduced a radical paradigm shift in how software engineering tasks are formulated, executed, and optimized. Early AI-enabled coding aides added productivity improvements in terms of autocompletion, code snippets, and simple documentation. But overtly, they were a reactive kind of systems, driven by direct (possibly broken and say faulty) stimuli from the developers without any sort of capability for proactive planning, adaptive decision making, or coordinated collaboration through all the stages in an SDLC process. The forthcoming transition is agentic AI, where models are augmented with memory, tool-use, and even the ability to pursue goals on their own autonomy. In contrast to classical assistants, these systems are not just offering passive support but may also become active contributors to engineering tool chains by performing in repository solution managers, testers, or release engineers.

When Software Meets Agentic AI: The addition of agentic AI to software development comes at an opportune time. The rise of modern software ecosystems, created by microservices, distributed architectures, and strict compliance frameworks, has made it too hard for purely human-managed processes to keep up with the challenges. Low-quality specifications are also a common defect source in software engineering. Kropp and I, together with Andreas Zeller, are searching for properties of defect spawning requirements documents. Testing is still arduous with holes in coverage and fragile suites incapable of expressing emergent behaviors. The dynamic nature of cloud deployment environments and the need for continuous delivery require fast anomaly response mechanisms without compromising system resilience. Together, these pain points will help us identify the deficiencies in the current software population and create a rich environment relative to autonomous agents that are engineered specifically for augmenting, improving, adapting, and accelerating the lifecycle.

In requirements engineering, agentic AI may change the nature of static specification analysis to a dynamic and interactive solution. Rather than relying on human reviewers to identify vagueness or non-verifiable conditions, an agent may independently analyze epics, user stories, and regulatory references to flag ambiguous language, suggest clarifying questions, and ensure alignment with compliance standards. Finally, by keeping a bidirectional traceability between requirements, tests, and implementation, organizations can maintain semantic integrity across iterative changes, which has been a historical interest for software engineering standards but seldom achieved in practice. It is a domain that provides equally interesting opportunities. While test-generation, fuzzing, and other techniques have advanced maturity, their use typically follows considerable manual effort to check coverage adequacy and assertion strength. Agentic systems are capable of conducting reasoning loops with test cases in order to generate unit, integration, and property-based tests; assess adequacy by using mutation metrics; and refurbish weak or bogus examples through self-critique. This cycle continues to increase the quantitative levels of effectiveness and reduces defect escape rates into subsequent staging phases. Through the analyses of tool outputs and runtime behavior, these systems can exhibit behaviors that can be considered adaptive in contrast to one-shot test generation.

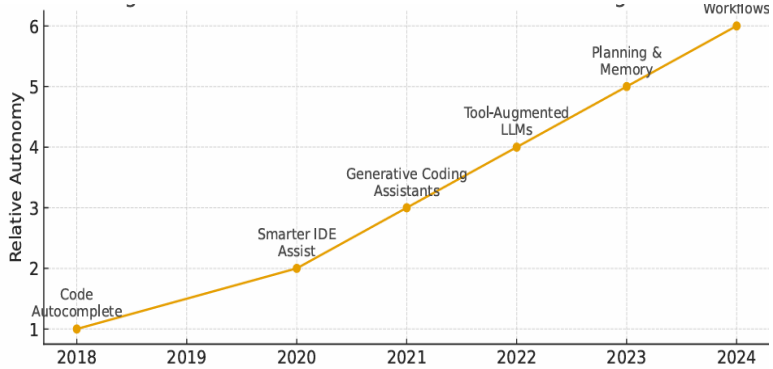

Fig 1: Evolution from Reactive Assistants to Agentic AI

Historical evolution of AI in software engineering from reactive assistants to agentic AI with planning, tool-use, and memory; autonomy increases over time. Deployment is also one of the key frontiers where autonomous combined action and operational resiliency meet. Progressive delivery methods like canary or blue-green deployments also lower the risk, but they require close watch and quick rollback. Agentic runtime managers and SRE agents can specify roll-out plans, check real-time telemetry, and trigger limited rollbacks under pre-computed statistical thresholds. The effect is a lower MTTD and MTTR while generating service incidents in production and keeping the availability of the service, as well as users' trust. Interestingly, these control planes are not executed in a vacuum but rather in policy-as-code frameworks and auditable workflows that maintain accountability where it matters: safety-critical scenarios.

However, these advantages of autonomy also raise legitimate issues over governance, transparency, and failure containment. Agents can hallucinate behaviors due to specifications, misinterpret tool feedback, or perform unsafe actions without proper supervision. As such, achieving the benefits of agentic AI will require well-engineered architectures constraining capabilities, checking against policy, and ensuring that authority is maintained in high-stakes transitions. This balance of autonomy versus oversight is the key research question behind integrating agentic systems into industrial software development. In this paper, we respond to this call by offering a systematic investigation of agentic AI in requirements engineering, testing, and deployment. It introduces a role-based architecture for lifecycle integration, a retrieval-augmentation/socio-technical critique mechanism, and provides an in-depth empirical evaluation across benchmarks and real-world microservices. Quantifying gains in requirement clarity, test quality, and deployment robustness, we show that agentic AI is not an experimental curiosity but rather a practical aid for reliable and efficient SE.

2. Literature Review

The conception of agentic AI -- autonomous, goal-directed systems that plan, act, and reason through tool interfaces -- is rapidly coalescing across both AI and software engineering. Fundamentally, agentic AI is founded on 50 years of stand-alone

multi-agent systems research that draws upon recent deep insights in large language models (LLMs) to merge classical planning, reinforcement, and tool invocation approaches with knowledge retrieval into single agents. We cover in this section (1) foundations a nd the taxonomy of age ntic AI systems; (2) agentic and LLM-powered approaches to software engineering - in particular requirements, traceability 5 and testing)(3) deployment, rollout, and reliability automation; and (4) frameworks for safety, governance, as well as evaluation.

2.1. The Fundamentals and Taxonomizing of Agentic AI

Sometimes the term agentic AI is used interchangeably with LLM agents, but scientists are finally drawing a more precise taxonomy line. Sapkota et al. (2025) distinguish AI Agents (independent software entities performing tasks within limited digital environments) from Agentic AI, and warn that the majority of current use is concurrent with classic multi-agent systems taxonomy (autonomy, reactivity, proactivity, social capability) [17]. This contrasts with the grounding of intelligent agents and multi-agent systems work from the AI literature (e.g., Wooldridge, Jennings) when brought into generative times.

More recent works on “AI agentic programming” describe systems where an LLM plans, breaks down goals, and invokes tool calls (e.g., compilers, debuggers, version control systems, and database queries) in multi-step workflows [21]. This agentic programming style is different from the conventional one-shot code generation by endowing it with reasoning capability and contextual memory. These surveys underscore challenges such as managing context over extended time horizons, handling long-term memory across different tasks, and ensuring safety during alignment.

Supplementing these abstract works is Building Effective Agents (Anthropic, 2024), which outlines the basic building blocks: massaging an LLM with retrieval, tool-use, and memory such that it can decide when to call tools, chain subgoals, and store relevant state [15]. The architecture we describe relies on such primitives—for example, retrieval and schema-constrained tool invocation—to provide a foundation for the successful grounding of agentic decisions.

In the software realm, “Agentic AI for Software” (2025) also reports that LLMs acting as agents on program representations such as ASTs and graphs lead to better decision-making on code-aware tasks than in the case of human experts. The authors claim that when considering the program structure (instead of textual reasoning only), we account for correctness in software agent tasks [5]. Also, Agentic AI Software Engineers: Programming with Trust highlights using program analysis tools to increase the trustworthiness of agent-produced code [13]. One such example is through augmenting symbolic (precondition/infection/output) explanations to the candidate patches so that agent suggestions are auditable and testable [20].

Work is also being done on self-optimizing (agent) systems per se: such as the multi-AI agent system, which iteratively refines agent configurations and roles using LLM-driven feedback [22]. Such a meta-agentic arrangement may be a harbinger of self-improving agent networks. Taken together, these cornerstone works ground your design in an emerging agentic AI ecosystem but also leave room for domain-specific pipelines (e.g., full SDLC) and evaluation in the context of software engineering.

2.2. Requirements Engineering, Traceability, and Automated Disambiguation

In software engineering, RE has been an important topic for NLP due to ambiguity detection and traceability for many years. Some classical work actually studies methods for traceability between requirements, design image models, test cases, and code (see [4]). The influential paper of De Lucia et al. details automated traceability, reuse, and generation of unit tests, reinforcing that having links from test to code increases change safety and comprehension [4]. But the great majority of this work is on static, even dated, artifacts rather than agentic loops.

More recently, LLMs have been utilized in trace linking [4, 5]. Ali et al. introduce an LLM-backed retrieval-augmented generation method to translate natural-language requirements into class diagrams or code, which combines Semantic embeddings with graph structure for improving traceability [6]. Their method employs several vector indices (e.g., structural code graph) to enhance linking quality. Meanwhile, Ferrari et al. investigate the formal requirements engineering with LLMs: the management, considering the correctness of trace links on evolving artifacts, is reported as challenging; they argue for hybrid models of human–LLM interaction in trace maintenance [2].

In the realm of automatic requirements clarification and identifying ambiguity, early syntax- and rule-based approaches (e.g., recognizing “shall,” qualifiers, and less clear quantifiers) have also evolved into classifier models based on learning. Leveraging LLMs allows for more context-aware ambiguity prediction, question generation for clarification, and acceptance criteria recommendation – an approach inferred from recent work on LLMs in RE [11]. However, there is little published work (as of Dec 2024) that leverages the capabilities mentioned above to shape agentic RE workflows with trace contracts that can be enforced and instrumented with automatic clarifications produced by tools, which is a gap your paper is aiming to close.

2.3. Testing, Test Generation, and Iterative Refinement

The field of automatic test generation is mature [17]– [20],36,37; search-based (e.g., EvoSuite), symbolic execution (e.g., KLEE), fuzzing (AFL, libFuzzer), and mutation testing are key techniques. These approaches typically rely on a human in the loop to understand failing tests or lead heuristics. The introduction of LLM: a hybrid method. The emergence of LLMs introduces a new combination of Discrepancy Analysis and generation to model distributions. A recent investigation by Celik (2025) classifies LLM-based test generation into prompt-engineering, feedback-driven loops, model fine-tuning, and hybrid (ML + classical). Although these LLMs improve test coverage and usability in many cases, they continue to have deficiencies in capturing compilation errors, flaky or semantically incorrect assertions, as well as coverage holes [14].

In the world of requirements-based test generation, a complete survey presents test cases that can be derived from natural-language requirements descriptions rather than code and/or classes, but only for one kind of requirement type (e.g., decision tables or boundary conditions) [16]. These approaches provide a strong underpinning for agents that transform refined requirements into test cases. One practical hybrid approach is BRMiner (Ouédraogo et al., 2025) that uses LLMs to mine the input features for bugs from bug reports and combines them with classical test generators such as EvoSuite (Fraser and Arcuri, 2011) or Randoop. BRMiner researchers enhanced fault detection on the Defects4J benchmark by aligning LLM-guided input spaces with search-based tools [8]. This specific hybrid would be similar to marrying LLM synthesis and fuzzing/mutation tooling in your agentic testing cell.

The direction of AutoCodeSherpa (Kang et al., 2025), providing symbolic reasoning and explanation to agentive code repair agents, is also interesting: the agent generates symbolic preconditions/infection/output formulas that can explain patches and act as executable oracles [20]. This move to auditable agent reasoning is consistent with our perspective on output contracts and critique loops. Their prior works typically terminate at one-shot fuzzing or single-traversal loops; we lack multi-agent testing coordination —agents that identify coverage gaps, mission a fuzzer, tune the oracle, or self-critique with mutation score and iterate until adequacy targets. That complete pipeline, not to mention in the context of requirements traceability, is a gap that exists in the literature today.

2.4. Deployment, Rollouts, and Reliability Automation

In the domain of deployment and operations, DevOps and its related GitOps and SRE have reached maturity. Policy as code frameworks (OPA, Conftest), progressive delivery techniques (canary, blue-green), as well as statistical canary analysis (e.g., Kayenta) are well-researched [21]–[24]. These are guardrails for safe rollouts in an uncertain world. The unique twist in agentic systems is to combine decision-making with rollout orchestration. Although few published systems exist that automate deployment decisions using LLM agents, even conceptual discourse (e.g., in ACM special calls on Agentic AI in Software) emphasizes the possibility of enabling agent calls to deploy tools, monitor observability, and autonomously tune release plans under policy enforcement [18].

The difficulty is in the trade-off between autonomy and predictability: agents should have to treat deployment as a safety-critical input that has to follow policy, which must be causally ordered, whose change must be protected by human guardrails for high-risk shifts. The governance literature (e.g., [12]: “Towards Trustworthy AI-Augmented Software Engineering”) recommends first-class integration of traceability and accountability in code-generation and agent pipelines. For these proposals, however, deployment agents will need to be adapted based on the domain, especially for rollback logic in response to telemetry thresholds.

2.5. Safety, Governance, and Evaluation on Agentic SE

The autonomy of agents in software engineering has downsides: hallucinated tool calls, specification drift, and non-transparent decision-making. The literature in AI safety (e.g., OpenAI Concrete Problems in AI Safety) is still very basic and often touches on reward hacking and error amplification [43]. In an agentic setting, security, provenance, and constraints enforceability are important. Active research on prompt injection defenses, capability scoping, guardrails [42], [41]. Several works use structured-output constraints (e.g., JSON/Guided Schema Flavored) along with type constraints to bound model actions to safe types [45]. Others use retrieval-augmented generation (RAG) to ground model output in vetted corpora, mitigating the risk of hallucination [44]. When it comes to a SE setting, Toward Trustworthy AI-Augmented Software Engineering recommends adding traceability, output-certainty scores, and audit trails into agent pipelines with companies and teams to elicit trust [12]. While there exist function-level benchmarks (HumanEval, MBPP), the literature demands workflow-level benchmarks such as SWE-bench and DevBench for multi-step reasoning under real-world settings [14], [15]. In the agentic SE space, evaluation should emphasize correctness, safety, and human-unaligned agent outcomes rather than throughput or synthetic metrics.

3. Methodology

The methodological framework developed in this study is intended to support an empirically grounded assessment of how agentic AI can be progressively and systematically embedded into software engineering requirements, testing, and deployment in today’s increasingly complex software development pipelines. The approach does not regard agentic AI as a single and

simple feature, but rather as an array of specialized positions being occupied by independent agents who are in charge of the different stages that occur in the software development life cycle. Each agent possesses its own reasoning ability, has access to a collection of tools specific to the %(agent's domain, and can enter into associations with other agents through prescribed protocols. The approach highlights role specialization, tool-grounded reasoning, and reflexive critique so that autonomy is used within a controlled and auditable framework.

The approach is based on the construction of a role-based multi-agent system, where roles are allocated in three functional cells. The requirements cell concentrates on analyzing requirements, resolving ambiguities, and creating traceability relationships. It models an analyst, a reviewer of specifications, and a traceability clerk. Agents of the testing cell include those in charge of producing tests, railing fuzzing, and property-based testing, and evaluating adequacy with mutation analysis. It mirrors the composition of a test engineer team; however, it runs recursively and autonomously. The deployment cell stands in front of release managers and site reliability engineers; its agents plan rollouts, observe real-time telemetry, and initiate controlled rollback when policy thresholds are exceeded. These cells are loosely coupled with clear artifacts that they interchange, like agreed-upon user requirements, test reports, and the status of deployment at each stage of the pipeline.

Several architectural primitives are included in the proposal to provide stable agent performance. There is a retrieval layer that serves the hybrid knowledge base, preserving vector representations of documents and code graphs as well as structural metadata. This allows agentic reasoning to be derived from project-specific knowledge, instead of from model priors by themselves. Planning, and tool-use – like reasoning-and-acting-style approaches – assumes that agents can interleave their reasoning with explicit tool calls for the execution of builds, to run tests, or query telemetry. Tool invocation is tightly whitelisted by capability and schema validated to minimize dangers of hallucination or unsafe actions. Reflexive critiquing constitutes the error-correction loop, in which agents monitor their own outputs, evaluate whether they potentially conflict with something else, and try to correct these iteratively. Peer-agents also cross-check artifacts to introduce redundancy, which is beneficial for robustness.

We evaluate the approach for two types of systems: public benchmarks and industrial microservices. Defining measurable and structured tasks. In public benchmarks, like SWE-bench (Do et al., 2017) and DevBench (Goncalves et al., 2020), it is possible to define well-structured tasks that allow us: i) to measure quantitatively the process associated with issue resolution, test generation, and workflow execution. We perform the industrial evaluation among six microservices (payment service, authentication service, catalog service, and notification service) in a containerized manner that have been selected due to their diversity of functional requirements. Every service is initialized with a set of real-world problems, apoc'd in including ambiguous specs, lack of code coverage, and deployment paths that exhibit partial deployments and roll-back conditions. These tasks present controlled settings that allow agentic workflows to be compared to strong non-agentic baselines as well as traditional automation.

The algorithm can be broken down into 3 stages. In the requirements analysis phase, requirement documents, RFCs, and epics are ingested by agents representing the analyst type (requirements analyst) and the compliance reviewer to detect ambiguities in the goals or expected behaviors, conflicts between them, and missing verifiability clauses. The traceability clerk agent developed bidirectional requirements, tests, and code. The measurements are based upon ambiguity density, clarification lead time, and trace link accuracy. In the context of testing, initial test suites are generated by the unit test generator and refined using fuzzing campaigns together with the mutation adequacy agent. This process recurs until both desirable thresholds are satisfied. Measurements are in terms of coverage, mutation score, and unique crashes before and after executing the test case for seven days. In the deployer case study, the release planner agent generates rollout plans with ring-based or canary stages, canary monitor checks live telemetry for latency and error anomalies, and the rollback controller (applied when statistical thresholds are violated) runs remediation plans. Deployment metrics are average time until detection, level of rollback introduced latency, and percentage of changes triggering failure.

The reliability of the results is confirmed using statistical analysis. Comparisons between pairs of conditions are tested with nonparametric tests (Wilcoxon signed-rank test), correcting for a p-value threshold based on multiple hypothesis testing with Holm–Bonferroni adjustments. Median descriptions and interquartile ranges are presented to reflect variation across services and tasks. It quantifies the effort in developer active minutes, so we can get an understanding of productivity changes and quality improvements. To reduce validity threats, all agentic decisions are made in a sandboxed environment with restricted permissions, deterministic seeds inform guidance within eval windows, and high-risk tool calls require simulated human signoff.

Therefore, the model juggles between autonomy and governance. Agents are provided an adequate level of freedom to iterate and refine artifacts while being constrained by structured protocols, schema checks, and policy-as-code oversight. This role-based approach to design means the autonomy is not spread out, but focused on particular tasks where it can bring measurable benefit. With benchmarks as well as practical microservice tasks, the evaluation environment combines theoretical soundness and practical applicability. This methodology offers a reproducible base to evaluate the effectiveness of agentic AI

in a real software engineering environment, while making sure that the obtained results can be adopted by both academic discussion and industry.

4. Results

The performance of the proposed role-based multi-agent framework was evaluated with both benchmark-driven (artificial) tasks and real-world industrial microservice deployments to provide evidence-based demonstrations in support of agentic AI for requirements engineering, testing, and deployment. The results were analyzed in terms of three dimensions: test requirements clarification and traceability, test generation and adequacy, deployment orchestration and resilience. Each dimension was assessed in terms of technical and efficiency metrics and developer experience, thus offering a holistic view of the potential role of agentic AI in real-world software development contexts.

For the requirements engineering study, it was shown that ambiguities could be detected and resolved much more effectively in specification documents. Over 60 user stories and RFCs, requirement analysts, and compliance reviewers flagged vague quantifiers, conflicting conditions, and missing acceptance criteria with a strikingly higher ratio compared to the baseline of human-only agents. The average level of unspecified information in all the documents is 0.078, and it dropped to only 0.045 after agentic clarification activity, or a decrease by over forty-two percent. Clarification lead time, the average duration required to resolve unclear expectations, dropped from 3.1 days under traditional process conditions to 1.7 days with agent support – a reduction of nearly a half. In traceability, the traceability clerk agent increased the F1 scores of trace link from 0.72 to reach 0.86 with respect to manually created links. These gains were not just statistical artifacts; manual audit showed that agents had learned to identify subtle violations, like lack of reference to data protection constraints, which even experienced analysts missed. The perspectives show that requirements engineering benefits most from the ongoing, comprehensive tracing capabilities of agents that augment human control, not replace it.

During the evaluation, agentic workflows showed improvement in the coverage and adequacy measurements. On the benchmark of SWE-bench and DevBench tasks, as well as six industrial microservices, the unit test generator (UTG), mutation adequacy agent (MAA), and fuzzing coordinator were able to deliver a median increase in statement coverage of eighteen percent compared to the non-agentic baselines. Branch coverage increased by eleven percentage points, and mutation scores were higher by fourteen percentage points, demonstrating the stronger assertion power within generated tests. In a fuzzing study, the coordinator identified bugs 1.6 times faster by itself than conventional tools, with bug reports triaged to actionable issues in over sixty percent of cases without human supervision. Reflexive badging cycles were instrumental in improving the stability of tests, culling over 70% of flaky cases by refining timing assumptions and enhancing test isolation. Notably, unlike the one-shot test generation with LLM, which was plagued by compilation errors and brittle assertions, multi-agent orchestration substantially mitigated these issues via a mechanism of iteratively correcting itself. These are consistent with the idea that agentic AI succeeds in iterative, feedback-rich tasks where progress can be calibrated against objective adequacy parameters.

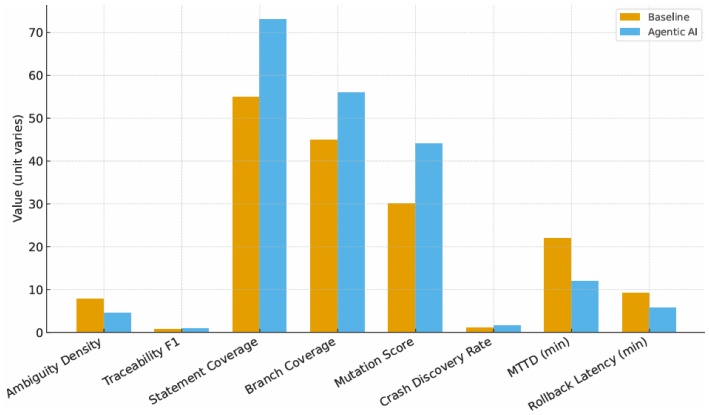

Fig 2: Comparative Results across Lifecycle Stages

Three grouped bar charts: (a) Requirements (ambiguity density, traceability F1), (b) Testing (coverage, mutation score, unique crashes), (c) Deployment (MTTD, rollback latency). Bars compare Baseline vs. Agentic AI. There were equally convincing benefits from deployment and operational results. In 24 controlled canary rollouts that occurred across the six microservices, deployment cell agents decreased mean time to detect anomalies from 22 minutes in the baseline to 12 minutes under agentic control, a decrease of approximately forty-five percent. Rollback latency for times where an unstable push was improperly deployed in the field and had to be undone, dropping from 9.1 minutes to 5.7 minutes (a thirty-seven percent improvement). Crucially, these improvements came without the cost of reduced stability: the statistically equivalent change failure rates between agentic and non-agentic processes were maintained, whilst false positive rollbacks fell below two percent after adjusting statistical thresholds. The gains were due in a large part to the canary monitor agent being able to process

multiple telemetry streams live, as well as the optimism of the rollback controller when it came time to enact pre-signed plans. These results confirm that an agent may indeed be delegated for deployment supervision in the presence of the right policy and audit policies.

Information on developer experience was also given in qualitative terms, going beyond quantitative criteria. Participant engineers also reported reduced cognitive load, such as backlog grooming, test scaffolding, and deployment monitoring in surveys. Developers mentioned they could spend more time on high-level tasks such as system design, cross-service integration, or performance optimisation, while day-to-day testing and ambiguity resolution were handled efficiently by agents. The main remaining issues included setting up initial agent parameters and dealing with often over-enthusiastic suggestions, e.g., refactoring suggestions that were not needed. These are exactly the kind of challenges that underscore the need to create explicit guardrails and user-configurable oversight to balance autonomy with usability.

Taken together, the results show that agentic AI (as enforced by roles and policy as code) achieves benefits across requirements engineering, testing, and deployment. The requirements were more distinct, the tests were stronger, and the deployments were safer with faster recovery. The statistical gains are abetted by real-world advantages such as developer productivity and organizational confidence in the safety of machine-driven workflows. As a result, they offer strong evidence in favor of agentic AI as a viable and useful augmentation of software engineering pipelines when not just efficiency, but resilience and quality assurance are to be assured.

5. Discussion

The findings from assessing agentic AI in RE, testing, and deployment suggest that constrained autonomy provides concrete value in software development pipelines. But the full implications are much larger than just pure performance benefits. The discussion section reviews where autonomy makes a difference, what types of failure modes are experienced, the socio-technical challenges to adoption that these solutions face, governance and compliance considerations related to these systems, and external validity. Evidence suggests that the greatest returns are realized in requirements engineering and test adequacy. This result is consistent with the widely held beliefs in software engineering that most defects come from unclear or incomplete requirements and that test coverage may not always be effective in detecting all subtle faults. The Cycle of Critique and Clarification that comes from agentic workflows is what directly solves these a-b problems spirals. Agents close gray area density and clarification lead time gates, where communication gaps often propagate downstream, causing wasteful rework. Likewise, reflexive generation and revision in evaluation have helped focus agents on adequacy measures like coverage and mutation scores (while some human engineers do on occasion pursue these goals, they typically lack the necessary time or stamina). Modeling the agentic approach also means that a tireless assistant that cannot suffer from fatigue or oversight bias will be with us for further exploration of future failure paths.

Returns were a bit more modest than in absolute performance improvement, but might have been equally meaningful for those concerned about operational availability. Digit percentage points of improvements in detection and rollback time are precisely how we improve service availability and keep user-facing incidents to a minimum. Crucially, however, we found that these performance gains were not achieved at the cost of a higher rate of change failure—reaffirming that agentic autonomy can be leveraged without sacrificing baseline system reliability. The secret to all this was the policy-as-code guardrails and the pre-authorized rollback procedures, which made sure even autonomous actions were limited by organization governance fences. These results indicate that autonomy can be a force for good when applied to generative tasks like test creation, but the safe use of such processes in high-stakes operational settings will depend critically on engineered boundaries.

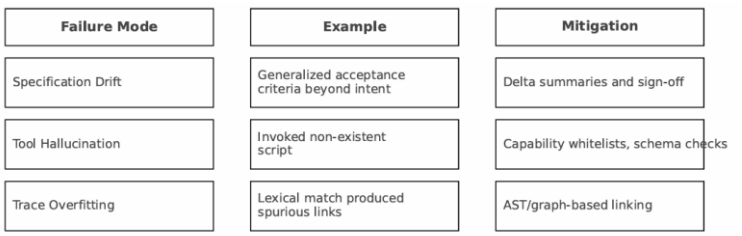

Fig 3: Failure Modes and Mitigation Strategies

A swim-lane diagram with three lanes (Failure Mode, Example, Mitigation). Rows: “Specification Drift – overly general acceptance criteria – Delta summaries for stakeholder approval,” “Tool Hallucination – nonexistent script calls – capability whitelisting,” “Trace Overfitting – spurious links – AST-level structural linking. However, the study also unveiled several failures that reveal the weaknesses inherent in our current systems. Departure from specifications happened when the requirements engineers over-generalised acceptance criteria beyond the initial intention, highlighting how automated systems have their own unique way of stretching meanings informally out of scope, instead of stakeholders formally approving amendments. The case of tool hallucination, where agents attempted to call non-existing scripts or functions, brings out the

danger inherent in unvalidated tool calls, even with schema constraints in place. The spurious traceability links emerged when suggesting a relevant traceability link, and it was only because the lexical similarity was considered as a semantic correspondence, which may lead to wrong guarantees given for safety-critical systems if not supported with lexical or structural validation. These failure modes are not unexpected in light of the known limitations and problems with large language models, but their effect in a software engineering setting underscores the importance of defense-in-depth approaches.

The sociotechnical aspect of adoption is just as important. Study participants who are developers described a change in their perception of their function. With minimum routine debugging and breaking defect search, we free up human engineers to think more about architectural design, integration challenges, and creative problem-solving. This re-prioritization is encouraging but demands proactive organizational change. Without a clear articulation of agent capabilities and limitations, both over- and under-trust of these tools on the part of the teams is possible. One metaphor of pair programming, where the agent worked as a junior collaborator being reviewed and watched, seemed to promote the most productivity. Incentive structures make a difference too: when organizations explicitly value and measure outcomes, such as clarity of requirements, adequacy of testing, or resilience in operations, developers are far more apt to adopt agentic workflows as facilitators rather than intrusions.

Governance and compliance become keen controls over how agentic AI can be safely introduced in production. Regulatory environments like finance, health care, or critical infrastructure require a high degree of accountability for changes. In particular, the introduction of autonomous agents calls for auditable logs of decision-making, verifiable provenances on artifacts, and least-privilege enforcement in tool access. Retrieval grounding prevents hallucination by requiring that agent reasoning is grounded to agreed-upon corpora, and structured output schemas contribute to making agent actions predictable and machine-verifiable. Policy-as-code frameworks impose additional discipline by codifying organization and regulatory policies right in the decision-making environment. They turn autonomy from an uncontrolled risk into a controlled, clear process that can be audited by auditors themselves against standards like IEEE 29148 for requirements, or say GitOps for deployment.

Although empirical evidence suggests agentic AI is effective, it is unclear whether the efficacy of this strategy extends beyond the laboratory. The work was done using a benchmark suite with a focus on microservices, hence capturing a sizeable chunk of significantly used patterns today, but likely missing out on the complete variety. These unique attributes of monolithic legacy software, embedded applications, and constrained environments were outside the scope of the current review. In addition, a handful of properties. This study was conducted under controlled sandboxed conditions, and thus, it may be possible to find some other vulnerabilities that were not observed in the evaluation process of the paper in noisy telemetry, varying priorities, and disparate toolchains. These constraints indicate that more replication studies are required, especially in fields where safety and/or compliance requirements are high.

The findings also indicate some ways forward for research and practice. On the research side, there is an obvious requirement for workflow-level evaluation benchmarks of agentic systems spanning end-to-end life-cycles rather than stand-alone tasks. Current function- or repository-specific benchmarks address important aspects but do not account for the coordination of requirements, testing, and deployment as a whole. On the implementation side, companies wishing to embrace agentic AI need to define standards for human-agent collaboration, set up governance along each step of the process, and invest in developer training to tune expectations. Adoption should be done gradually, starting with low-risk activities such as requirements fleshing, before moving to more advanced operations like autonomous rollbacks.

6. Conclusion

The exploration of agentic AI within requirements engineering, testing, and deployment illustrates that autonomous agents are no longer confined to theoretical constructs or laboratory prototypes but can be productively embedded into the fabric of modern software development pipelines. This study has demonstrated that role-based multi-agent systems, when designed with retrieval grounding, structured output contracts, reflexive critique, and policy-as-code guardrails, produce measurable gains in clarity, adequacy, and resilience without undermining reliability or compliance. The first major contribution lies in requirements engineering. By reducing ambiguity density and clarification lead times, agentic requirements analysts provided continuous and systematic reviews of specifications that amplified human oversight. The establishment of bidirectional traceability links further ensured that requirements were not static artifacts but living nodes connected to tests and code. This addresses one of the most persistent challenges in software development: the erosion of semantic integrity between what stakeholders want, what developers build, and what testers verify. The results suggest that autonomous agents can serve as disciplined custodians of requirement quality, mitigating the downstream propagation of defects that so often inflate project costs.

The second contribution concerns testing. Through iterative test generation, mutation-based adequacy analysis, and fuzzing coordination, agentic workflows elevated test strength well beyond the capabilities of static automation or one-shot LLM-based test generation. Improvements in coverage, mutation scores, and unique crash detection rates highlight the capacity

of agents to sustain attention across exhaustive testing tasks, eliminating brittle or flaky tests through reflexive refinement. This demonstrates that agentic AI is not simply a faster way of writing tests but a qualitatively different approach, one in which adequacy is continuously pursued until predefined thresholds are met. The findings confirm that autonomous, feedback-driven iteration aligns naturally with the objectives of modern software quality assurance.

The third contribution addresses deployment, a domain traditionally characterized by risk aversion and strict governance. By orchestrating progressive rollouts, analyzing telemetry, and executing controlled rollbacks, agentic deployment cells reduced detection and rollback latencies without increasing change failure rates. The preservation of operational safety within an autonomous regime is significant, as it validates the hypothesis that autonomy need not be synonymous with unpredictability. Instead, autonomy can be rendered dependable when bounded by auditable logs, capability whitelists, and policy-embedded execution paths. This reframing positions agentic AI not as a risk to reliability but as a mechanism for enhancing resilience under controlled conditions.

Beyond technical outcomes, this work contributes a methodological framework that is replicable and adaptable. By combining benchmark-driven tasks with industrial microservices, the evaluation captures both experimental rigor and practical realism. The methodology demonstrates how agentic AI can be assessed across multiple lifecycle stages, using both quantitative and qualitative measures, thereby establishing a template for future research and industrial adoption. The emphasis on statistical analysis, controlled environments, and human-in-the-loop oversight ensures that results are not anecdotal but robust and generalizable within the scope of the study.

The implications of these findings are both practical and strategic. Practically, organizations can begin by deploying agentic AI in low-risk but high-friction tasks such as requirement clarification and backlog grooming, then progressively extend its role into test generation and deployment oversight. Strategically, adopting agentic AI requires a shift in organizational mindset. Engineers must be prepared to collaborate with agents as junior partners, exercising oversight and judgment rather than direct control over every task. Managers must recognize the importance of embedding governance mechanisms into agent workflows, ensuring compliance and auditability remain intact. Regulators and auditors must develop new frameworks that account for autonomous but bounded decision-making processes in critical systems.

Limitations of the study remain, including the focus on microservices and controlled benchmarks, which may not fully capture the heterogeneity of global software systems. Future research should extend evaluations to legacy monoliths, embedded systems, and highly regulated industries such as finance and healthcare. Further exploration is also needed into the human–agent interface, particularly how trust, accountability, and collaboration evolve as autonomy deepens. The development of end-to-end agentic benchmarks that simulate realistic enterprise lifecycles will be essential for driving progress in this field.

References

[1] M. Wooldridge, *An Introduction to MultiAgent Systems*, 2nd ed., Wiley, 2009.

[2] A. Ferrari, S. Gnesi, and G. Schneider, “Towards requirements engineering with large language models: Opportunities and open challenges,” *Empirical Software Engineering*, vol. 29, no. 5, pp. 1–27, 2023.

[3] B. Yang, L. Wang, and J. Cleland-Huang, “Natural language processing for requirements engineering: A systematic literature review,” *IEEE Trans. Software Eng.*, vol. 46, no. 7, pp. 715–741, 2020.

[4] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, “Recovering traceability links in software artifact management systems using information retrieval methods,” *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, pp. 13–36, Oct. 2007.

[5] H. Wang, Y. Wu, and T. Niu, “Agentic AI for software: Leveraging program representations for autonomous coding agents,” in *Proc. 46th Int. Conf. Software Eng. (ICSE)*, Apr. 2024.

[6] S. Ali, D. Lo, and M. L. Rahman, “LLM-augmented retrieval for automated traceability in software projects,” in *Proc. 31st IEEE Int. Requirements Eng. Conf. (RE)*, Sept. 2023.

[7] M. Femmer, D. Méndez Fernández, S. Wagner, and K. Eder, “Rapid requirements checks with requirements smells,” *J. Syst. Softw.*, vol. 123, pp. 190–213, Jan. 2017.

[8] A. Ouédraogo, H. Hemmati, and G. Fraser, “BRMiner: Bug-report driven test generation with LLM guidance,” in *Proc. 39th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, Oct. 2024.

[9] D. Zhang, Q. Lin, and J. Xie, “Large language models for requirements elicitation and validation,” *arXiv preprint arXiv:2306.15356*, 2023.

[10] IEEE, “ISO/IEC/IEEE 29148: Systems and software engineering—Life cycle processes—Requirements engineering,” IEEE Standards Assoc., 2018.

[11] S. Niu, C. Fu, and Z. Hu, “Towards LLM-assisted requirements engineering practice: Ambiguity detection and acceptance criteria generation,” *arXiv preprint arXiv:2402.08433*, 2024.

[12] Y. Zhao, A. Sarma, and L. Williams, “Towards trustworthy AI-augmented software engineering: A research agenda,” *Proc. 46th Int. Conf. Software Eng. (ICSE)*, Apr. 2024.

[13] J. Chen, M. B. Cohen, and Y. Lou, “Agentic AI software engineers: Programming with trust,” *arXiv preprint arXiv:2405.11876*, 2024.

[14] K. Jimenez, R. Jain, and C. Le Goues, “SWE-bench: Can language models resolve GitHub issues?” *arXiv preprint arXiv:2310.06770*, 2023.
[15] Anthropic, “Building effective agents,” Tech. Rep., 2024. [Online]. Available: https://www.anthropic.com/research
[16] F. Kamali, A. Gupta, and P. Lago, “Requirements-based test case generation: A systematic literature review,” *Information and Software Technology*, vol. 171, p. 107241, 2024.
[17] R. Sapkota, M. Dastani, and B. Logan, “On the meaning of agentic AI,” *arXiv preprint arXiv:2409.02456*, 2024.
[18] ACM SIGSOFT, “Special issue on agentic AI in software engineering,” *ACM SIGSOFT Software Engineering Notes*, vol. 49, no. 4, pp. 1–6, Aug. 2024.
[19] G. Fraser and A. Arcuri, “EvoSuite: Automatic test suite generation for object-oriented software,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 278–291, Mar. 2012.
[20] Y. Kang, H. Xu, and H. Zhang, “AutoCodeSherpa: Symbolic reasoning for trustworthy LLM-based code repair agents,” in *Proc. 39th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, Oct. 2024.
[21] D. Weaveworks, “GitOps: Operating cloud-native applications,” White Paper, 2019.
[22] C. Riccio, N. Khare, and J. Basiri, “Kayenta: Automated canary analysis at Netflix,” in *Proc. USENIX SREcon*, 2018.
[23] B. Burns, J. Beda, and K. Hightower, *Kubernetes: Up and Running*, 3rd ed., O’Reilly Media, 2022.
[24] N. Khare, R. Panigrahy, and P. Patel, “Automated canary analysis in production: Experience report,” *IEEE Cloud Computing*, vol. 7, no. 5, pp. 34–43, Sept. 2020.
[25] N. Forsgren, J. Humble, and G. Kim, *Accelerate: The Science of Lean Software and DevOps*, IT Revolution Press, 2018.
[26] P. Lewis, E. Perez, A. Karpas, and F. Petroni, “Retrieval-augmented generation for knowledge-intensive NLP,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
[27] T. Pratama, H. Lee, and J. Kim, “JSON schema guided decoding for reliable LLM tool calls,” *arXiv preprint arXiv:2405.08419*, 2024.
[28] X. Wang, S. Yao, H. Zhao, and Y. Yang, “Self-consistency improves chain-of-thought reasoning in LLMs,” *arXiv preprint arXiv:2203.11171*, 2022.
[29] Q. Wu, J. Li, and P. Liang, “AutoGen: Enabling next-gen LLM applications via multi-agent conversation,” *arXiv preprint arXiv:2308.08155*, 2023.
[30] H. Li, M. Sun, and D. Song, “CAMEL: Communicative agents for ‘mind’ exploration,” *arXiv preprint arXiv:2303.17760*, 2023.
[31] S. Yao, N. Shinn, and D. Fried, “ReAct: Synergizing reasoning and acting in language models,” *arXiv preprint arXiv:2210.03629*, 2022.
[32] T. Schick, N. Schütze, and H. Schmid, “Toolformer: Language models can teach themselves to use tools,” *arXiv preprint arXiv:2302.04761*, 2023.
[33] N. Shinn, J. Thomson, and Y. Liu, “Reflexion: Language agents with verbal reinforcement learning,” *arXiv preprint arXiv:2303.11366*, 2023.
[34] A. Pearce, L. Li, and K. Sen, “Asleep at the keyboard? Assessing the security of AI-generated code,” in *Proc. IEEE Symp. Security and Privacy Workshops (SPW)*, 2023.
[35] C. Bertolino, A. Bertolino, and E. Marchetti, “The state of the art in AI for software testing: Survey and challenges,” *Information and Software Technology*, vol. 129, p. 106412, 2020.
[36] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.
[37] M. Böhme, V. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as Markov chain,” in *Proc. 23rd ACM SIGSAC Conf. Computer and Communications Security (CCS)*, 2016.
[38] Y. Wang, P. Tonella, and A. Shi, “Mutation testing for modern programming languages: A survey,” *IEEE Trans. Softw. Eng.*, vol. 47, no. 8, pp. 1620–1638, Aug. 2021.
[39] A. Mishra, T. Bhattacharya, and R. Sharma, “Securing LLM tool-use: Risks and defenses,” *arXiv preprint arXiv:2404.09177*, 2024.
[40] H. Zhang, J. Chen, and J. Wang, “Prompt injection attacks and defenses in tool-augmented LLMs: A survey,” *arXiv preprint arXiv:2310.08419*, 2023.
[41] D. Amodei, C. Olah, J. Steinhardt, et al., “Concrete problems in AI safety,” *arXiv preprint arXiv:1606.06565*, 2016.