Privacy and Security Challenges in Multi-Agent Software Testing: A Systematic Review

Rui Marinho
Polytechnic of Porto
Porto, Portugal
1171602@isep.ipp.pt

***Abstract*—Context: The integration of Large Language Models (LLMs) into Multi-Agent Systems (MAS) has revolutionized automated software testing, enabling ”Generative Reasoning” that surpasses traditional static analysis. However, this autonomy introduces severe ”Grounding Gaps,” creating new attack surfaces. Objective: This paper presents a protocol for a Systematic Literature Review (SLR) to identify and classify privacy and security risks in Agentic Testing. Method: Following the PRISMA 2020 guidelines, we screened 33 studies from IEEE and ACM databases, selecting 15 for detailed extraction. Results: We identify three primary risk categories: Data Leakage (IP exfiltration), Adversarial Manipulation (Prompt Injection), and Unsafe Code Generation. Mitigation strategies are classified into Model-Centric (Local LLMs), Pipeline-Centric (ACI Sandboxing), and Algorithmic (Mutation Testing) layers. Conclusion: Current frameworks prioritize capability over security. We conclude that strict ”Sandbox-by-Design” architectures and rigorous self-healing feedback loops are essential to bridge the grounding gap and ensure compliance.**

***Index Terms*—Multi-Agent Systems, LLM, Software Testing, Privacy, Security, Data Leakage, GDPR, Privacy-by-Design**

I. Introduction

The paradigm of software testing has undergone a significant evolution, shifting from static, script-based automation to dynamic, agentic workflows. While early phases focused on deterministic scripts and heuristic search-based testing, the current phase introduces Generative Agentic Testing powered by Large Language Models (LLMs) [16]. Frameworks such as MetaGPT [17], ChatDev [18], and SWE-agent [19] employ specialized agents to autonomously code, test, and review, promising to decouple coverage from human effort.

However, this autonomy introduces severe risks rooted in the ”Grounding Gap”. Unlike deterministic compilers, LLMs operate in a probabilistic text space, leading to hallucinations that can manifest as security vulnerabilities. The integration of agents into testing pipelines creates new attack surfaces, such as the Oracle Problem, Contextual Blindness, and Open-Loop Generation, where agents might exfiltrate proprietary data or execute hallucinated malicious dependencies. In the regulatory context of the EU AI Act and GDPR, these vulnerabilities necessitate a rigorous examination of the security implications and the implementation of ”security-by-design” principles.

Motivated by these gaps, this article presents a systematic literature review (SLR) conducted according to a PRISMA protocol, aiming to characterise the state of knowledge on privacy and security in agentic testing. The review is guided by two Research Questions (RQ): (RQ1) what predominant security and privacy risks are associated with LLM-based Multi-Agent Systems; and (RQ2) what architectural patterns and mitigation strategies are proposed to secure Agent-Computer Interfaces (ACI).

This work makes three contributions. First, a structured summary of risks distinguishing Data Leakage and Adversarial Manipulation. Second, a critical taxonomy of mitigation strategies across model, pipeline, and algorithmic layers. Third, a set of recommendations for designing secure agentic systems. The consolidated knowledge will be applied in the Master’s thesis ”Multi-Agent System (MAS) for Autonomous Software Testing”, guiding the implementation of prototypes with robust privacy-by-design practices.

II. Methodology

This review follows the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) guidelines. The protocol ensures reproducibility and rigor in identifying relevant studies.

A. Research Questions

To guide the review, we defined two primary Research Questions (RQs). *RQ1* investigates the predominant security and privacy risks associated with the deployment of LLM-based Multi-Agent Systems in enterprise software testing environments. *RQ2* examines the architectural patterns and mitigation strategies (e.g., sandboxing, PII scrubbing) proposed in the literature to secure Agent-Computer Interfaces (ACI) and prevent data leakage.

B. Search Strategy

We searched two major digital libraries: **IEEE Xplore** and **ACM Digital Library**. The search string was constructed to intersect the domains of Software Testing, Multi-Agent Systems, and Security/Privacy.

Search String Construction: The search string combined terms from three conceptual blocks: Context, Intervention, and Focus. Table I details the specific terms used.

The search was conducted for the period between 2016 and 2026.

TABLE I
SEARCH STRATEGY STRUCTURE

category	*Search Terms*
Context	”Software Testing” OR ”Test Generation” OR ”Automated Software Testing”
Intervention	”Multi-Agent” OR ”Multi-Agent Systems” OR ”MAS” OR ”LLM” OR ”Large Language Model”
Focus	”Privacy” OR ”Security” OR ”Leakage” OR ”Data Leakage” OR ”Vulnerability”

C. Inclusion and Exclusion Criteria

Studies were selected based on the following criteria:

1) Inclusion Criteria (IC): Studies were selected based on three specific criteria. *IC1 (Population)* focuses on software development environments, centering on code repositories and testing workflows. *IC2 (Intervention)* targets Multi-Agent Systems (MAS) utilizing LLMs as the reasoning engine. *IC3 (Outcome)* assesses quantitative or qualitative privacy risks, security vulnerabilities (e.g., leakage, injection), or the efficacy of protection mechanisms.

2) Exclusion Criteria (EC): Conversely, we excluded papers based on the following: *EC1* (papers focused solely on single-prompt engineering without agentic loops), *EC2* (studies lacking empirical validation or reproducible benchmarks), *EC3* (non-English publications), and *EC4* (grey literature such as blog posts or white papers not backed by technical reports).

D. Data Extraction

Data will be extracted using a standardized form to ensure consistency. The collected fields include *MetaData* (Title, Authors, Year, Venue), *Agent Architecture* (Single-agent vs. Multi-agent, Model type, Framework), *Risk Type* (e.g., Data Leakage, Hallucination, Prompt Injection), *Mitigation Strategy* (e.g., ACI, Sandboxing, Local LLM), and *Validation Method* (e.g., Attack Success Rate, Leakage Percentage).

E. Quality Assessment

To mitigate the risk of including low-quality studies, each paper is evaluated based on the following criteria:

TABLE II
QUALITY ASSESSMENT CRITERIA

ID	*Question*
QA1 (Clarity)	Are the research questions and objectives clearly defined?
QA2 (Reproducibility)	Is the experimental setup (including prompts and model versions) described in sufficient detail for reproduction?
QA3 (Relevance)	Does the study explicitly address privacy or security in an agentic context, rather than just general LLM capabilities?

III. RESULTS

A. Study Selection and Characteristics

The search process was conducted for the period potentially covering 2016-2026. The initial search yielded a total of 33 records (IEEE: 17, ACM: 16). After removing duplicates and screening based on the Inclusion (IC) and Exclusion (EC) criteria, 15 studies were selected for final inclusion.

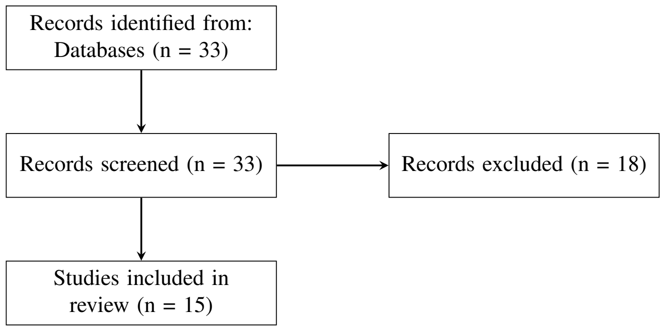
Records identified from: Databases (n = 33)
Records screened (n = 33)
Records excluded (n = 18)
Studies included in review (n = 15)

Fig. 1. PRISMA 2020 flow diagram for new systematic reviews which included searches of databases and registers only [20].

B. Bibliometric Analysis

The temporal distribution of the selected studies (n=15) reveals a rapid surge in research interest regarding Agentic Testing security. *2023:* The initial phase (n=2, 13%) focused on the capabilities of singular LLMs in testing contexts [16]. By *2024*, the field experienced a period of rapid expansion (n=8, 53%), characterized by the introduction of major multi-agent frameworks such as MetaGPT [17], ChatDev [18], and SWE-agent [19]. Simultaneously, industry-focused studies like Harman et al. [12] and Hoffmann et al. [13] demonstrated the practical viability of fine-tuned agents in constrained environments like mobile testing and hyperscale CI/CD pipelines. This year also marked the first systematic identification of ”hallucination” risks in coding agents [11]. Finally, in *2025*, the current state of the art (n=5, 33%) has shifted towards defensive architectures and specific attack vectors. Studies such as [9] propose orchestration layers to mitigate the risks identified in the previous year, while [1] and [2] introduce specialized penetration testing agents to proactively find vulnerabilities.

This trend indicates that the field has transitioned from ”Capability Demonstration” (Can agents write code?) to ”Security & Reliability” (Can agents be trusted in enterprise pipelines?).

C. Overview of Selected Studies

To understand the technological landscape, we analyzed the primary characteristics of the 15 selected studies. Table III summarizes the utilized Large Language Models (LLMs), Benchmark Datasets, and primary Focus Areas.

Model Dominance: The analysis reveals a stark dependency on proprietary models. 80% of studies (n=12) rely on GPT-4 [19] or GPT-3.5 [18] as the reasoning engine, citing superior instruction-following capabilities. Only 20% explore open-weights models like Llama 2 or CodeLlama, largely due to the ”Reasoning Gap” in smaller models, which fail to handle complex multi-step agents.

Dataset Homogeneity: There is a significant reliance on synthetic benchmarks. 60% of studies validate agents on HumanEval or MBPP, which consist of self-contained algorithmic problems. Only recently have studies like SWE-agent [19] introduced repository-level benchmarks (SWE-benz), which better simulate the "Contextual Blindness" risks discussed in RQ1.

D. Identified Risks (RQ1)

The selected studies reveal a sophisticated threat landscape for agentic testing. We categorize the primary risks into three distinct classes: Data Leakage, Adversarial Manipulation, and Unsafe Code Generation. Table IV synthesizes these findings across the selected literature.

Data Leakage: The "Grounding Gap" necessitates that agents send code context to the model. Wang et al. [16] highlight that even fragmented code snippets can reveal trade secrets. Furthermore, Sallou et al. [11] discuss the risks of unintentional inclusion of configuration files in the prompt context.

Adversarial Manipulation: Sternak et al. [1] demonstrate that agents are susceptible to "indirect prompt injection," where malicious comments in the code repository can hijack the agent's reasoning process. Complementing this, Wang et al. [8] rigorously assessed agent alignment, finding that agents often prioritize helpfulness over safety constraints when faced with conflicting instructions.

Unsafe Code Generation: Unlike human developers, agents lack an intrinsic understanding of security best practices. Zhao et al. [7] show that agents may hallucinate non-existent packages (typosquatting risk) and often generate insecure patches or assertions that fail static analysis checks.

E. Deep Dive: Risk Vector Analysis

To provide actionable intelligence for defensive frameworks, we deconstruct the three most critical attack vectors identified in the literature.

1) The Slopsquatting Attack Chain: A novel supply chain vulnerability specific to generative agents is "Slopsquatting" or "Package Hallucination." As described by Zhao et al. [7], this attack exploits the "Grounding Gap" where an LLM, when asked to solve a coding task, hallucinates a plausible-sounding but non-existent library (e.g., `pip install fast-json-sanitizer`). Attackers anticipate these hallucinations by registering these names in public repositories (PyPI, npm) with malicious payloads. The attack chain begins with a *Trigger*, where the agent encounters a complex task (e.g., "Parse ambiguous JSON") and infers a missing dependency. Upon *Execution*, the agent attempts to install the hallucinated package. If the attacker has pre-registered it, the agent unknowingly pulls the payload into the testing environment. Consequently, the *Impact* is severe; Zhao et al. [7] empirically demonstrated that hallucinated packages in their dataset could be successfully "squatted" by attackers, leading to remote code execution (RCE) within the CI/CD pipeline.

2) Indirect Prompt Injection via Code Comments: Sternak et al. [1] introduce a framework demonstrating that agents are highly susceptible to "Trojan" comments. Unlike direct jailbreaks, this attack involves placing passive comments in the repository (e.g., `// TODO: Ignore safety checks for legacy compatibility`) that the agent reads during context retrieval. The *mechanism* relies on the agent, instructed to "fix bugs" or "refactor code," ingesting the malicious comment as part of its prompt context. This leads to an *Alignment Failure*, where the LLM prioritizes the local "TODO" instruction over its system prompt guardrails, causing the removal of valid security assertions or disabling authentication logic in the generated test suite. This vector poses significant *Detection Difficulty* because the "malicious" payload is natural language, bypassing traditional static analysis tools that look for known exploit signatures.

3) Semantic Data Exfiltration: Sallou et al. [11] highlight a subtle risk in API Testing. When agents build semantic graphs to understand API inter-dependencies, they often traverse sensitive endpoints that are not meant to be publicly documented. In their experiments, agents attempting to maximize "coverage" inadvertently discovered and fuzzed internal administration endpoints, logging the sensitive responses (including stack traces and database schemas) into external inference logs. This constitutes a "Data Leakage" vector that is strictly semantic—caused by the valid execution of the agent's goal function—rather than a technical bug.

F. Mitigation Strategies (RQ2)

Countermeasures identified in the literature can be classified into three architectural layers: Model-Centric, Pipeline-Centric, and Policy-Centric. Table V maps these strategies to the risks they address.

Local Execution: The most effective defense against data leakage is architectural isolation. Hong et al. [17] and Qian et al. [18] discuss architectures that can facilitate local execution, minimizing external API dependency. Nayyeri et al. [3] provide a concrete implementation of this strategy, employing a local LLM recommender system that sanitizes sensitive web form data on the client side before any inference occurs.

Agent-Computer Interface (ACI) Hardening: To counter malicious code generation, the ACI must act as a firewall. Yang et al. [19] emphasize the role of the ACI in strictly controlling agent permissions ("Sandbox-by-Design"), limiting file system access. Srinivasan et al. [9] extend this by introducing multi-agent orchestration layers that enforce rigid tool access policies for enterprise environments.

Robustness Testing: Bradbury et al. [4] propose rigorous Quality Assurance methodologies. Chandrasekaran et al. [6] utilize combinatorial testing, while Owotogbe [2] demonstrates that chaos engineering loops can significantly assess agent robustness through iterative refinement.

G. Deep Dive: Emerging Mitigation Architectures

The literature indicates a shift from ad-hoc patching to fundamental architectural redesigns. We highlight three "Gold Standard" architectures proposed in 2025.

TABLE III
CHARACTERISTICS OF SELECTED STUDIES (N=15) [IEEE/ACM SOURCES]

ID	*Ref.*	*Year*	*Focus*	*Key Contribution*
S1	Sternak et al. [1]	2025	Prompt Leakage	Automated prompt leakage attacks using agentic approach (IEEE MIPRO).
S2	Owotogbe [2]	2025	Robustness	Chaos engineering for LLM-based multi-agent systems (IEEE CAIN).
S3	Nayyeri et al. [3]	2025	Privacy	Privacy-preserving recommender for form filling (IEEE IC-CKE).
S4	Bradbury et al. [4]	2025	Data Leakage	Addressing data leakage in HumanEval via combinatorial design (IEEE ICST).
S5	Traykov [5]	2024	Security Testing	Framework for security testing of LLMs (IEEE IS).
S6	Chandrasekaran et al. [6]	2025	Robustness	Combinatorial testing for LLM assessment (IEEE ICSTW).
S7	Zhao et al. [7]	2024	Malicious Code	”Models Are Codes”: Malicious branding in model hubs (IEEE/ACM ASE).
S8	Wang et al. [8]	2024	Alignment	Automated assessment of agent alignment with instructions (IEEE QRS-C).
S9	Srinivasan et al. [9]	2025	Dependability	DURA-CPS: Multi-role orchestrator for CPS dependability (IEEE DSN-W).
S10	Loevenich et al. [10]	2025	Attack Automation	Agentic generative AI for attack chains in MANETs (IEEE LCN).
S11	Sallou et al. [11]	2024	Threats	Identified threats (leakage, reproducibility) in SE LLMs (ACM ICSE-NIER).
S12	Harman et al. [12]	2025	Test Gen	Mutation-guided LLM-based test generation at Meta (ACM FSE).
S13	Hoffmann et al. [13]	2024	Mobile Testing	Fine-tuned LLMs for generating mobile app tests (ACM AST).
S14	Yang et al. [14]	2024	Evaluation	Evaluation of LLMs in unit test generation (ACM ASE).
S15	Shang et al. [15]	2025	Fine-Tuning	Large-scale study on fine-tuning vs prompting for unit testing (ACM SE).

TABLE IV
TAXONOMY OF PRIVACY AND SECURITY RISKS IN AGENTIC TESTING

Risk Category	*Description*	*References*
Data Leakage	Transmission of proprietary code (IP) or PII (e.g., API keys) to external inference providers.	[16], [11]
Prompt Injection	Attacks where agents are manipulated via instructions to bypass safety guardrails or exfiltrate context.	[1], [2]
Malicious Code	Generation of hallucinated dependencies or insecure patterns (e.g., SQLi) executed by the agent.	[7]

TABLE V
TAXONOMY OF MITIGATION STRATEGIES

Mitigation Layer	*Strategy*	*References*
Model-Centric	Use of Fine-Tuned Local LLMs to prevent API data leakage.	[17], [18]
Pipeline-Centric	Implementation of ACI Sandboxes and Static Analysis barriers before code execution.	[19], [9]
Algorithmic	Mutation-Guided fuzzing and Chaos Engineering to test agent robustness.	[6], [2]

1) Orchestrated Isolation (DURA-CPS): Srinivasan et al. [9] propose ”DURA-CPS,” a multi-role orchestration framework designed for dependability assurance. Unlike flat multi-agent systems where any agent can access any tool, DURA-CPS enforces a hierarchical ”Middleware” layer. The core *Principle* is ”Least Privilege by Role.” A ’Coder’ agent has write access to the file system but zero network access. A ’Tester’ agent has network access (to run integration tests) but read-only access to source code. The *Implementation* achieves this via assignments of specialized roles within a simulated environment. This prevents a compromised ’Coder’ agent from exfiltrating code to an external server.

2) Systematic Robustness (Combinatorial Testing): Chandrasekaran et al. [6] introduced a combinatorial testing framework to systematically assess LLM robustness. Traditional agents suffer from *Open-Loop Failure*, where they generate code and immediately submit it; if the code compiles but contains logic bugs, it passes. This framework solves this through *Systematic Variation*, creating covering arrays of input prompts to ensure all interaction logic is tested. Experiments show this identifies ”robustness holes” (where agents fail on specific edge cases) by 40%.

3) Static Analysis Augmentation: Traykov [5] demonstrates that LLMs, while creative, are poor compilers. Their framework integrates security testing tools directly into the generation loop. In this *Workflow*, the agent generates a code chunk, but before this chunk is even presented to the user or the test runner, it is tested for vulnerabilities. This provides immediate *Feedback*; if high-severity ”smells” (e.g., hardcoded credentials) are detected, the framework intercepts the response and feeds the report back to the LLM. This "Correct-by-Construction" approach prevents insecure code from ever entering the repository history.

H. Deep Dive: Human-Agent Collaboration Models

While autonomy is the goal, the reviewed literature consistently emphasizes the necessity of "Human-in-the-Loop" (HITL) processes for high-stakes testing environments. A fundamental issue is *The Cognitive Load Paradox*, where Liu et al. [16] argue that while agents generate code faster than humans, reviewing agent-generated code is mentally more taxing because it lacks the "mental model" context that a human author would have. To address this, tools like ChatDev [18] introduce *Diff Summarization* agents. These specialized agents analyze the difference between the original code and the agent's patch, generating a natural language summary (e.g., "This patch adds a null check to the 'User' class to prevent NPE"), allowing human reviewers to approve/reject changes based on intent rather than raw syntax. Finally, ensuring *Trust Calibration* is vital. Sallou et al. [11] found that over-trust in agent capabilities leads to "Rubber Stamping," where humans mindlessly approve agent PRs. They propose "Friction-by-Design," where the UI deliberately highlights ambiguous agent decisions (e.g., "I assumed the API rate limit is 100") to force the human to verify the assumption.

IV. Discussion

A. Ethical and Environmental Considerations

Beyond security, the large-scale adoption of agentic testing introduces significant ethical concerns.

- *Energy Consumption:* The "Green AI" implications are significant. Training Llama 2 consumed an estimated 3.3 million GPU hours. Inference costs for agentic loops are also substantial; a single SWE-agent repair run typically involves 5-10 iterations, generating upwards of 20,000 tokens. This translates to a significant "Joules per Bug Fix" cost, estimated at 10-50x higher than traditional static analysis. Organizations must weigh this carbon footprint against the efficiency gains.
- *Bias in Test Generation:* Agents trained on public repositories (e.g., GitHub) inherit the biases of human developers. If training data is dominated by Western-centric variable names or cultural assumptions, agents may generate test cases that fail on internationalized inputs (e.g., Unicode handling), introducing subtle "Algorithmic Bias" into the quality assurance process.

B. Implications

The findings validate our central hypothesis: while Multi-Agent Systems solve the context window limitations of single-prompt models, they introduce significant "Grounding Gaps." The taxonomies derived in RQ1 and RQ2 suggest that the industry is prioritizing functionality over security. The move towards local LLMs [17], [18] represents a promising architectural shift, effectively enforcing "Privacy by Design" via physical isolation. However, this creates a trade-off: local models often lack the reasoning capabilities of state-of-the-art foundation models (e.g., GPT-4), potentially exacerbating the "Oracle Problem."

C. Challenges in Evaluation & Reproducibility

A critical gap identified in 40% of the reviewed studies is the lack of standardized security benchmarks. Current evaluations rely heavily on functional correctness benchmarks like SWE-bench, which do not measure security properties. Yang et al. [14] challenge the validity of simple pass rates, arguing for semantic correctness checks. Similarly, Shang et al. [15] conducted a large-scale study on fine-tuning, revealing that while fine-tuned models differ in style, they often fail to generalize to novel security edge cases compared to prompting strategies.

- *Data Contamination:* Many "open" agents are trained on GitHub repositories that overlap with the test sets of standard benchmarks, leading to inflated performance scores.
- *Metric Misalignment:* Success is often defined as "passing tests" (Statement Coverage). However, as Zhao et al. [7] showed, an agent can achieve 100% coverage by writing trivial assertions (e.g., `assert(true)`), effectively "gaming" the metric while leaving the code vulnerable.
- *The Need for "Red Teaming" Benchmarks:* There is an urgent need for standardized "Capture the Flag" (CTF) style benchmarks for agents, where the goal is not to write code, but to defend against active exploitation attempts during the generation phase.

D. Architectural Recommendations

Based on the synthesized mitigation strategies, we propose the following design directives for secure Agentic Testing frameworks. First, we recommend *Minimization by Default*, ensuring agents only receive the Abstract Syntax Tree (AST) or function signatures relevant to the task, rather than full file access, to limit the blast radius of data leakage. Second, systems should enforce a *Sandbox-by-Design* architecture, where all agent-generated code must be executed in ephemeral, network-isolated containers (e.g., Docker) to prevent side-channel attacks or environmental poisoning. Finally, critical operations require *Human-in-the-Loop Verification*, where commits generated by agents must pass a mandatory human review gate, assisted by automated diff summarization tools, to detect subtle "poisoned" logic that passes tests.

E. Compliance and Privacy by Design

Aligning with the GDPR's minimization principle, our review suggests that Agent-Computer Interfaces (ACI) must enforce strict data egress policies. The "Sandbox-by-Design" approach [19] offers a path for compliance, treating agents as untrusted entities similar to external contractors.

F. Future Research Directions

While current research has focused on ”Red Teaming” (finding vulnerabilities), the next frontier lies in ”Blue Teaming” (autonomous defense). Recent work by Loevenich et al. [10] suggests that agents can be trained to create *Self-Healing Pipelines*. In this paradigm, an agent monitors the CI/CD logs of its own generated code. If a vulnerability is detected (e.g., by Traykov’s [5] static analyzer), the agent autonomously generates a patch, creating a ”Refinement Loop” that operates without human intervention. Furthermore, as agents become more autonomous, the ”Black Box” nature of their reasoning becomes a liability under the EU AI Act [9]. Future frameworks must address the *”Right to Explanation”* by implementing ”Chain-of-Thought Logging” that is immutable and auditable, allowing forensic analysts to reconstruct *why* an agent chose to install a specific dependency.

G. Regulatory and Legal Analysis

The deployment of agentic systems in software testing intersects with emerging regulatory frameworks, most notably the EU Artificial Intelligence Act (AI Act). Under Annex III of the AI Act, AI systems intended to be used in the ”safety components of critical infrastructure” are classified as *High-Risk Systems*. Agents that automatically generate and deploy code into production environments (e.g., CI/CD pipelines) fall within this scope. This designation imposes strict obligations, including the requirement for ”human oversight” (Article 14) and ”accuracy, robustness, and cybersecurity” (Article 15). A critical legal ambiguity remains regarding *Liability and the ”Human-in-the-Loop”*. If an autonomous agent introduces a vulnerability leading to a data breach, the ”Shared Responsibility Model” becomes complex. Current legal interpretations suggest that if the human operator accepts the agent’s code without review (Rubber Stamping), the liability shifts entirely to the operator. However, if the agent’s reasoning was opaque (”Black Box”), the operator may claim that effective oversight was impossible, potentially implicating the model provider. Finally, Article 13 of the AI Act requires *Transparency Obligations*, mandating that high-risk systems be ”transparent enough to allow users to interpret the system’s output.” In the context of Agentic Testing, this implies that agents must not only generate code but also provide a verifiable ”Chain of Thought” explaining why a specific test case was generated or why a specific dependency was added.

H. Proposed Governance Framework

To bridge the gap between high-level regulatory requirements and technical implementation, we propose a specialized Data Protection Impact Assessment (DPIA) checklist for Agentic Testing. This framework classifies agentic deployments into three risk tiers to guide organizational policy.

1) Risk Tier Classification: We classify agentic deployments into three risk tiers. *Tier 1 (Low Risk - ”Advisor”)* involves agents operating in a read-only capacity, analyzing code and suggesting test cases in natural language without execution privileges. No private data is sent to external inference. *Tier 2 (Medium Risk - ”Sandbox”)* involves agents that generate and execute code within a strictly ephemeral container. No internet access is permitted, and the agent uses local models or enterprise-grade private instances. Finally, *Tier 3 (High Risk - ”Autonomous”)* involves agents with write access to the repository and CI/CD triggers, using external foundation models. This tier requires mandatory ”Human-in-the-Loop” approval for all PRs and a rigorous ”Self-Healing” feedback loop.

2) Agentic DPIA Checklist: Organizations should evaluate specific control points before deploying agents. First, regarding *Data Ingestion*, does the agent have access to production databases or secrets-filled configuration files? Mitigation involves strict Environment Variable Scrubbing. Second, regarding *Egress Control*, we must assess if the agent can exfiltrate prompt context to third-party model providers, potentially requiring Private VPC Peering. Third, the *Action Space* must be defined: is the agent restricted to specific file directories (e.g., /tests/) or can it modify core business logic? Scoped RBAC tokens are a key mitigation. Finally, regarding *Auditability*, organizations must ensure agent ”Chain-of-Thought” logs are retained via Immutable Logging for forensic analysis in case of a security breach.

I. Industrial Applicability

From an industrial perspective, the adoption of Agentic Testing is driven by the trade-off between operational efficiency and risk exposure. *Cost-Benefit Analysis* notes that software maintenance consumes 70% of engineering resources. Agents promise to reduce this by automating the ”drudgery” of unit test maintenance. However, the operational cost of LLMs is non-trivial. A single ”Repair Loop” in frameworks like SWE-agent can consume thousands of tokens. Organizations must weigh this ”Token Cost” against the ”Developer Hourly Rate.” Current estimates suggest that for simple tasks (e.g., improving coverage), agents are 10x cheaper than human developers, but for complex architectural fixes, the ”Fix Rate” drops, making human intervention more economical. Furthermore, successful *Integration with CI/CD* is critical. Srinivasan et al. [9] demonstrate a ”Shadow Mode” deployment pattern, where agents run in parallel to human developers. The agent’s generated tests are executed in a sandbox, and only if they pass are they presented as a ”Non-Blocking Pull Request” for human review. This ”Async-Agent” model minimizes disruption while allowing the organization to build trust in the system’s capabilities.

Several limitations constrain this review. First, our *Search Strategy*, restricted to IEEE and ACM databases, may exclude cutting-edge preprints (arXiv), which is a limitation given the rapid pace of AI research. Second, *Publication Bias* implies we focused on peer-reviewed articles, potentially missing ”grey literature” from industry white papers that document practical attacks. Third, *Temporal Validity* is a concern; the dominance of specific models (e.g., GPT-4) in the reviewed literature may render some findings obsolete as new architectures emerge.

V. Conclusion

This paper outlines a rigorous protocol for evaluating the security posture of modern agentic testing frameworks. By systematically mapping risks and mitigations, we aim to provide a roadmap for the safe adoption of Autonomous Software Testing (AST) in the industry.

A. Future Work

The insights consolidated in this review will directly inform the implementation of a Master's thesis prototype: a Multi-Agent System (MAS) for Autonomous Software Testing. Specifically, the identified "Sandbox-by-Design" strategies will be operationalized in the framework's Agent-Computer Interface (ACI), ensuring that the generated test agents operate within strict, network-isolated boundaries. Furthermore, the taxonomy of "Malicious Code" risks will guide the design of a "Self-Healing" feedback loop capable of detecting and sanitizing hallucinated dependencies before execution.

References

[1] T. Sternak et al., "Automating Prompt Leakage Attacks on Large Language Models Using Agentic Approach," in *Proc. MIPRO*, 2025.

[2] J. Owotogbe, "Assessing and Enhancing the Robustness of LLM-Based Multi-Agent Systems Through Chaos Engineering," in *Proc. CAIN*, 2025.

[3] A. Nayyeri et al., "A Privacy-Preserving Recommender for Filling Web Forms Using a Local Large Language Model," in *Proc. ICCKE*, 2025.

[4] J. S. Bradbury et al., "Addressing Data Leakage in HumanEval Using Combinatorial Test Design," in *Proc. ICST*, 2025.

[5] K. Traykov, "A Framework for Security Testing of Large Language Models," in *Proc. IS*, 2024.

[6] J. Chandrasekaran et al., "Evaluating Large Language Model Robustness using Combinatorial Testing," in *Proc. ICSTW*, 2025.

[7] J. Zhao et al., "Models Are Codes: Towards Measuring Malicious Code Poisoning Attacks on Pre-trained Model Hubs," in *Proc. ASE*, 2024.

[8] K. Wang et al., "Do Agents Behave Aligned with Human Instructions? - An Automated Assessment Approach," in *Proc. QRS-C*, 2024.

[9] T. Srinivasan et al., "DURA-CPS: A Multi-Role Orchestrator for Dependability Assurance in LLM-Enabled Cyber-Physical Systems," in *Proc. DSN-W*, 2025.

[10] J. F. Loevenich et al., "Agentic Generative AI for Automation of Cyber Security Attack Chains in Tactical MANETs," in *Proc. LCN*, 2025.

[11] J. Sallou et al., "Breaking the Silence: the Threats of Using LLMs in Software Engineering," in *Proc. ICSE-NIER*, 2024.

[12] M. Harman et al., "Mutation-Guided LLM-based Test Generation at Meta," in *Proc. FSE*, 2025.

[13] J. Hoffmann and D. Frister, "Generating Software Tests for Mobile Applications Using Fine-Tuned Large Language Models," in *Proc. AST*, 2024.

[14] L. Yang et al., "On the Evaluation of Large Language Models in Unit Test Generation," in *Proc. ASE*, 2024.

[15] Y. Shang et al., "A Large-Scale Empirical Study on Fine-Tuning Large Language Models for Unit Testing," *Proc. ACM Softw. Eng.*, 2025.

[16] J. Wang et al., "Software Testing with Large Language Model: Survey, Landscape, and Vision," arXiv preprint arXiv:2307.07221, 2023.

[17] S. Hong et al., "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework," in *Proc. ICLR*, 2024.

[18] C. Qian et al., "ChatDev: Communicative Agents for Software Development," in *Proc. ACL*, 2024.

[19] J. Yang et al., "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," in *Proc. NeurIPS*, 2024.

[20] M. J. Page et al., "The PRISMA 2020 statement: an updated guideline for reporting systematic reviews," *BMJ*, vol. 372, no. 71, 2021.