Automated Generation of Microservice Authorization Tests Using Large Language Models

Md Arfan Uddin*, Shakthi Weerasinghe*, Connor Wojtak*, Tomas Cerny*
*Department of Electrical and Computer Engineering
University of Arizona, Tucson, USA
Email: {arfan, syweerasinghe, connorwojtak, tcerny}@arizona.edu

Deuslirio Silva-Junior‡, Mateus Eduardo S. Ribeiro‡, and Manoel Veríssimo dos Santos Neto‡
‡Institute of Informatics
Federal University of Goiás, Goiânia, Brazil
Email: {deuslirio_junior, mateus.eduardo, manoelverissimo}@discente.ufg.br

Valdemar V. Graciano-Neto‡, and Arlindo Galvão‡
‡Institute of Informatics
Federal University of Goiás, Goiânia, Brazil
Email: {valdemarneto, arlindogalvao}@ufg.br

Amr S. Abdelfattah†
†Meta Inc., California, USA
Email: amrs@meta.com

***Abstract*—Microservice architectures are inherently plagued by "authorization blindspots"–divergent security policies across independent services that create undetectable downstream security drifts. As systems evolve, these invisible vulnerabilities leave applications highly susceptible to privilege escalation and catastrophic data breaches. To eliminate these blindspots, we introduce a novel, fully automated framework that bridges the precision of formal static analysis with the adaptiveness of Generative AI. By extracting a policy-enriched Intermediate Representation of the microservice system, our approach deterministically guides GPT-5 to synthesize executable, downstream-aware policy test suites targeting specific policy inconsistencies. Evaluation on the Train-Ticket benchmark denotes that our method outperforms state-of-the-art tools such as EvoMaster and EvoSuite by generating 100% semantically valid authorization policy tests. Further, this research provides vital empirical validation for formal methods. By producing 97.4% error-free drift validation tests, our approach systematically neutralizes static analysis noise. Ultimately, these results establish a rigorous, highly effective pathway for hybridizing formal structures with Large Language Models to definitively verify complex, distributed authorization policies.**

***Index Terms*—Automated Test Generation, Authorization Testing, Microservices, Large Language Models, Authorization Drift, Static Analysis.**

I. Introduction: The Authorization Blind Spot

While microservices significantly enhance organizational agility and system scalability [1], their decentralized nature inherently fragments the enforcement of authorization policies. Since a single user request typically traverses multiple interacting services, each enforcing access control locally, a structural gap arises between global security intentions and their localized implementations. This fragmentation generates complex policy environments where subtle inter-service interactions can yield critical vulnerabilities [2]. Existing testing paradigms struggle to robustly address these distributed risks. Notable high-stake data breaches involving Google+ [3] and Uber [4] originated from distributed authorization failures, illustrating why "Broken Access Control" consistently ranks as the primary web application security risk by OWASP [5]. Consequently, contemporary automated testing tools such as EvoSuite [6], EvoMaster [7], RESTler [8], RESTTestGen [9], and LlamaRestTest [10] are not equipped to close this gap, resulting in persistent "authorization blindspots" (Fig. 1). These tools predominantly generate *shallow* tests that explicitly target the entry-point API layer while neglecting deeper, distributed execution pathways and cross-service policy variations [11]. Without downstream awareness, state-of-the-art approaches lack the context necessary to formulate meaningful assertions for detecting authorization drift.

Alternative methodologies also exhibit significant limitations: manual testing lacks the scalability required for microservices [12], and formal verification [13] typically fails to produce executable test artifacts. While Large Language Models (LLMs) demonstrate robust capabilities for test generation [14], applying them directly is hindered by context window constraints, code noise, and an inability to natively interpret distributed architectural logic. Building upon inter-procedural control flow graphs (ICFG) [15], we posit that call graphs augmented with authorization metadata offer an optimal foun-

* All experiments, data collection, and data processing activities reported in this study were conducted by the university collaborators and on university-managed resources. Meta participated solely in an advisory capacity. No experiments, data collection, or data processing activities were conducted on Meta infrastructure.

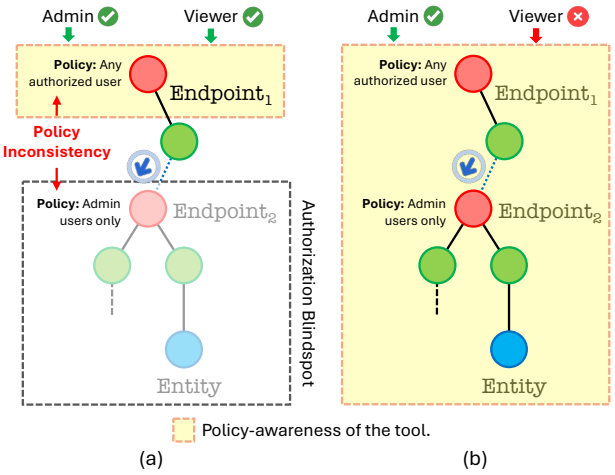

Fig. 1. For a given testing scenario, (a) the shallow policy awareness of existing tools overlooks inconsistencies and yields false-positives; whereas (b) downstream policy awareness correctly identifies true-positives and true-negatives.

dation for driving LLM-based, downstream policy-aware test generation to effectively identify security drift [16].

In this paper, we introduce a novel, automated framework designed to systematically identify authorization drift in microservice architectures while providing empirical validation for formal methods. Our approach abstracts systems into a policy-enriched intermediate representation (IR), utilizing binary authorization vectors at each tier of the call graph. By deriving scenarios that encompass all potential authorization outcomes, we leverage LLMs with adaptive token provisioning to synthesize executable tests. In contrast to existing solutions, our framework strictly preserves downstream awareness. We advance the state of microservice authorization testing through four key contributions: (1) proposing a downstream policy-aware framework capable of systematic drift detection; (2) providing empirical validation for formal drift detection via the generation of executable test artifacts; (3) introducing a scalable methodology that surpasses existing benchmarks for pipeline integration; and (4) conducting a comprehensive analysis of LLM-generated tests regarding their validity, quality, and sanitization.

These contributions are validated against the Train-Ticket benchmark, where our framework achieved 100% authorization scenario coverage, yielded 97.4% runnable authorization verifications, and generated 100% precise natural language explanations.

The remainder of this paper is structured as follows: Section 2 examines the specific challenges of microservice authorization and discusses the limitations of current techniques. Section 3 outlines our proposed methodology. Section 4 presents the empirical results of our case study. Section 5 explores lessons learned, broader implications, and ongoing research challenges. Finally, Section 6 concludes the paper and suggests avenues for future work.

II. From Microservice Testing to LLM-Generated Code

The shift from monolithic systems to microservices necessitates a fundamental reassessment of authorization (AuthZ) verification. In monolithic environments, security is governed by centralized, localized kernels. In contrast, distributed architectures rely on interprocedural communications that cross network boundaries, transforming AuthZ from static access control lists into dynamic, distributed execution properties. Consequently, security validation can no longer rely on isolated checkpoints; it demands comprehensive, end-to-end verification.

A. The Evolution and Fragmentation of Authorization

The dispersion of enforcement points across a microservice system introduces several compounding security challenges. Firstly, distributed AuthZ decisions demand intricate inter-service coordination, often relying on context that may be incomplete or stale. The number and complexity of these interactions frequently surpass the diagnostic limits of conventional testing frameworks [16], [17]. Further, systems often suffer from policy fragmentation. Different microservices may employ heterogeneous access control models such as Role-Based Access Control (RBAC) at the perimeter and Attribute-Based Access Control (ABAC) internally, obfuscating the verification of global security invariants [18].

Since microservices maintain independent deployment lifecycles, they are susceptible to *authorization drift*—the phenomenon where deployed code gradually deviates from formal security specifications [13], [19]. Drift is particularly common during rapid refactoring phases. For example, if a refactored `CheckoutService` calls an internal `PaymentService` without properly propagating the user's authorization token, it inadvertently introduces a confused deputy vulnerability. This risk landscape is further complicated by the use of service meshes [19], whose infrastructure-level configurations can conflict with application-layer security logic, generating a vast state space that is difficult to test without semantic awareness of the underlying AuthZ mechanisms [20].

B. The Verification Gap

Existing verification methodologies typically offer either high-level theoretical proofs or superficial runtime coverage, leaving a critical gap in actionable security testing. Manual testing approaches are fundamentally unscalable for modern microservices [12]. Furthermore, even when test suites achieve high statement coverage, they frequently exhibit low mutation scores, indicating that critical security logic remains unexercised [21].

Existing Search-Based Software Testing (SBST) tools falter in distributed environments. Tools including RESTTestGen [9] and RESTler [8] treat backend services opaquely, ignoring vital inter-service dependencies. EvoSuite [6] and EvoMaster [7] similarly restricts its analysis to shallow API entry points [11]. Even MISH [11], which attempts to capture sequential dependencies using dynamic execution logs, relies

on a reactive, grey-box methodology that may struggle to effectively traverse the expansive AuthZ search spaces present in moderately sized systems.

Conversely, formal methods (FM) based policy verification provide mathematical rigor but suffer from a distinct lack of empirical validation [13]. They cannot conclusively guarantee that a running implementation (subject to environmental variables and sidecar proxies) actually enforces the theoretically proven properties.

C. Leveraging Large Language Models for Security Testing

Large Language Models (LLMs) introduce a level of semantic awareness noticeably absent in traditional SBST frameworks [14]. They have demonstrated significant versatility in assessing test adequacy [21], fuzzing cyber-physical web applications [20], and uncovering complex API vulnerabilities such as Broken Object Level Authorization [22], [23].

Nevertheless, deploying LLMs for distributed AuthZ testing presents substantial hurdles. *The Oracle Problem* often leads models to generate superficial assertions (e.g., validating only an HTTP `200 OK` status) rather than deeply inspecting structural security invariants, often necessitating elaborate zero-shot Chain-of-Thought prompting [24]. Moreover, empirical studies highlight a severe execution gap: system tests generated by LLMs frequently require up to 29% manual remediation due to hallucinated dependencies and improperly formatted authentication headers [17]. Even heavily fine-tuned models, e.g., LlamaRestTest [10], are constrained to resolving single-endpoint dependencies derived from OpenAPI specifications [23].

As outlined in Table I, existing LLM-based frameworks primarily target monolithic applications or isolated APIs, relying on inputs that inherently lack the structural rigor necessary to trace inter-service AuthZ chains. However, an LLM's inherent ability to deduce security intent [25] implies that they can be highly effective at detecting drift *if* structurally anchored within a rigid architectural context (Fig. 2).

D. Bridging the Gap: A Policy-Centric Approach

Our review demonstrates that while AuthZ has matured into a complex, distributed execution property, the tools used to verify it remain heavily fragmented. The resulting "authorization blindspot" is characterized by three distinct research gaps: (1) SBST frameworks prioritize structural code coverage over actual security semantics [21]; (2) Formal methods supply theoretical assurances but lack the empirical grounding necessary for live runtime validation [26]; and (3) LLMs possess semantic capabilities but are hindered by weak oracles, unable to systematically validate deep policy invariants without a structured representation of the distributed system [10], [22].

To reconcile formal mathematical rigor with empirical execution, we propose a scalable, policy-aware methodology. By leveraging a policy-enriched Intermediate Representation (IR), we ground LLMs directly in formal security predicates. This approach transforms abstract security policies into actionable, empirical evidence, effectively closing the loop between theoretical specifications and actual runtime enforcement. We investigate the efficacy of this approach by addressing the following Research Questions (RQs):

- **RQ1:** Can Large Language Models be effectively guided by policy-enriched representations to synthesize downstream-aware authorization tests capable of navigating complex microservice call chains?
- **RQ2:** How effective are these LLM-generated security oracles and test logic in providing concrete empirical validation for authorization drift?

III. From IR to Executable Test Cases

To render downstream authorization behavior fully testable, our methodology systematically models the transitive service-to-service call chains where security drift and policy conflicts typically manifest. By expanding upon graph-grounded LLM prompting techniques [15], this automated pipeline integrates explicit authorization modeling, deep downstream reasoning, and adaptive test execution.

Figure 2 illustrates the end-to-end workflow: beginning with Java Spring Boot source code, the system generates a policy-enriched Intermediate Representation (IR), extracts an Endpoint Behavior Graph (EBG) to map cross-service dependencies, calculates chain-aware authorization matrices, and ultimately synthesizes executable testing scenarios.

A. Software Architecture Reconstruction (SAR)

We leverage a Software Architecture Reconstruction (SAR) paradigm [2], [27] to build a static IR that yields two vital perspectives: (1) an authorization policy map that binds user roles to specific endpoints, and (2) endpoint schemas necessary for precise request formulation. This framework-aware, lightweight snapshot acts as the foundational source of truth for all generated artifacts.

Targeting Java Spring Boot (a popular microservice framework) to validate our language-agnostic approach, we utilize JavaParser for the static extraction of REST endpoints. This process captures routing details, HTTP methods, parameters, and structural body schemas. To construct the **authorization matrix**, we hierarchically resolve distributed security policies: global rules (e.g., `SecurityFilterChain`) are superseded by class-level annotations, which are in turn overridden by fine-grained method-level annotations (e.g., `@PreAuthorize`).

This resolution produces an authorization metadata tuple for every endpoint e, formalized as $AuthZ(e) = \langle R_{allow}, R_{deny}, isPublic \rangle$. Here, R_{allow} represents permitted roles, R_{deny} specifies restricted roles, and $isPublic$ signifies unauthenticated accessibility. Fusing this metadata with the system model elevates a standard control-flow graph into a policy-enriched IR, equipping it to systematically reason about downstream security drift.

B. Endpoint Behavior Graphs (EBG)

Serving as the **structural backbone** of our analysis, the EBG correlates initial entry requests with the cascade of downstream endpoints they trigger.

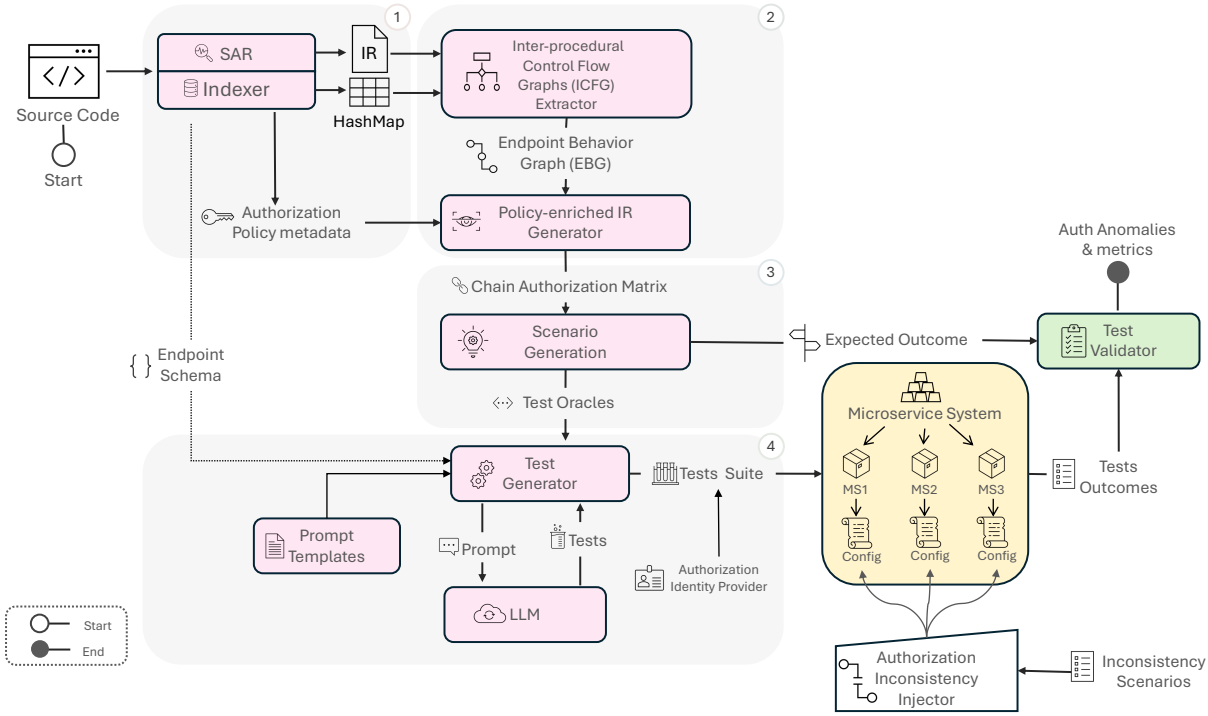

Fig. 2. Downstream authorization-aware endpoint testing in microservices.

TABLE I
COMPARISON OF RECENT LLM-ASSISTED TESTING METHODOLOGIES.

Testing Paradigm	Target Domain	Input Context Source	Primary Objective	Execution Reliability
API Security and Fuzzing [10], [20], [22], [23]	REST APIs, PLCs	OpenAPI Specs, Firmware State Machines	BOLA, AuthN, Inter-parameter dependencies	Low (Relies on mock data)
Test Amplification and Oracles [24], [25]	REST APIs	Existing Tests, Dynamic Execution Logs	Bug Detection, Invariant Filtering	Medium
System Testing [17]	Web Applications	Natural Language Requirements	E2E Functional Scenarios	Low (Requires manual tweaks)
Proposed Framework	**Distributed Microservices**	**Formal Static Analysis (Policy IR)**	**Authorization Drift Validation**	**High (100% Executable)**

1) Indexing and Stable Identifiers: To facilitate highly scalable graph analysis, all extracted program elements are assigned stable SHA-256 identifiers. These are linked to constant-time lookup structures via canonical signatures (outlined in Table II). This methodology guarantees compact graph artifacts and streamlines cross-component linking.

By hashing each canonical string into a fixed-length identifier, the system efficiently maps references to their corresponding structural records, seamlessly bridging Control Flow Graph (CFG) generation and EBG assembly.

2) Per-Method CFG Construction: An *EnhancedICFG* (Figure 3) is generated for every method, mapping out its loops, branches, and call sites. Local invocations are statically resolved through JavaParser. Conversely, remote invocations are identified by analyzing HTTP clients (e.g., `RestTemplate`), extracting URL templates to generate a precise `targetedEndpointId`.

3) EBG Composition and Considerations: These isolated, per-method CFGs are then woven into the comprehensive, endpoint-centric *EBG*. The stitching process initiates at the controller handlers and propagates outward via call-site linkages: `targetedMethodId` connects local methods, while `targetedEndpointId` maps to remote services. This architecture explicitly highlights cross-service transitions while retaining control-flow integrity (illustrated by the orange edges in Figure 3). Empirical observations indicate that highly dynamic URL generation obscures less than 5% of remote calls; to avoid inaccurate linkages, these are safely maintained as unresolved nodes. Infinite recursion is prevented through a strict visited-set policy.

C. Downstream AuthZ Analyzer

Utilizing the EBG and the authorization policy map, our framework calculates a chain-aware authorization matrix to expose access discrepancies that remain hidden at the initial entry point. For a defined set of global roles $\mathcal{R}$, the metadata of endpoint e is converted into a binary vector $\mathbf{a}(e) \in \{0,1\}^{|\mathcal{R}|}$, where $\mathbf{a}(e)[r] = 1$ signifies that role r is permitted. For any given entry endpoint e_0 traversing an EBG call chain $\pi = \langle e_0, e_1, \ldots, e_k \rangle$, the holistic, chain-aware authorization is

TABLE II
COMPONENT INDEX FORMATS

Component	Full Index Format	Examples (from TrainTicket)
Class	`{microserviceName}:{packageName}.{className} #class:{classType}:{classPath}`	`ts-user-service:com.example.service. UserService#class: CLASS:/src/.../UserService.java`
Method	`{microserviceName}:{packageName}.{className} #method:{static|instance}#method: {methodName|<init>}({paramTypes}):{returnType}`	`ts-user-service:com.example.service. UserService#method:instance#method: getUserById(String,Long):User`
Field	`{microserviceName}:{packageName}.{className} #field:{static|instance}#field:{fieldName}: {fieldType}`	`ts-user-service:com.example.service. UserService#field:instance#field: userRepository:UserRepository`
Annotation	`{packageName}.{className}#annotation: {annotationName}({key1}={value1},...)`	`com.example.service.UserService#annotation: RequestMapping(value=/api/users,method=GET)`
Parameter	`{packageName}.{className}.{methodName}. {paramName}:{paramType}`	`com.example.service.UserService.getUserById. userId:String`
MethodCall	`{packageName}.{className}#call:{objectName}. {methodName}({parameterContents}): {sequenceNumber}`	`com.example.service.UserService#call: userRepository.findById(userId):0`
Import	`{static:}{importPackage}.{importObject}`	`static:java.util.Collections.emptyList`
Endpoint	`{normalizedService}:{normalizedUrl}: {httpMethod}`	`userservice:/api/v1/users/{id}:GET`

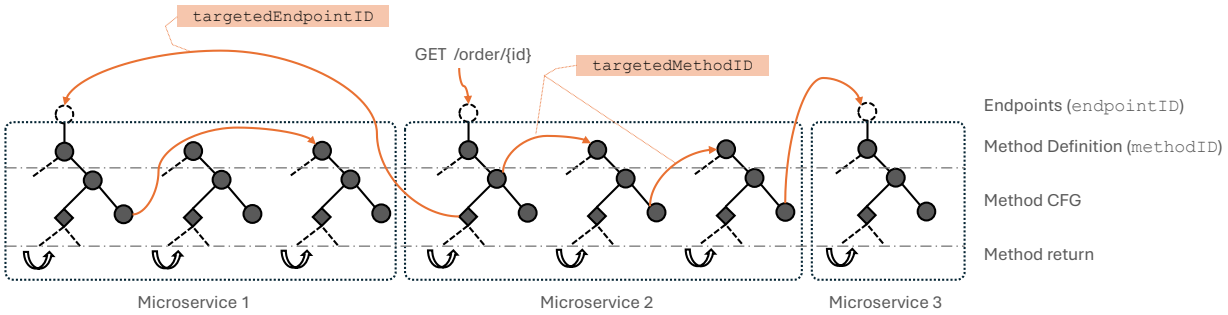

Fig. 3. Representative Control Flow Graph (CFG) within a distributed microservice system.

derived via the boolean conjunction across the entire pathway:

$$\mathbf{a}^{\text{chain}}(\pi) = \mathbf{a}(e_0) \wedge \mathbf{a}(e_1) \wedge \cdots \wedge \mathbf{a}(e_k). \tag{1}$$

Fundamentally, a request role is only viable for chain π if it successfully passes the authorization checks at every sequential endpoint.

Example. Assume $\mathcal{R} = \{\texttt{ADMIN}, \texttt{USER}\}$, ordered as $[\texttt{ADMIN}, \texttt{USER}]$. Consider an entry point e_0 that triggers e_1, which subsequently calls e_2, forming $\pi = \langle e_0, e_1, e_2 \rangle$. If the individual endpoint vectors are established as:

$$\mathbf{a}(e_0) = [1, 1], \quad \mathbf{a}(e_1) = [1, 1], \quad \mathbf{a}(e_2) = [1, 0].$$

The resulting chain-aware vector calculates to:

$$\mathbf{a}^{chain}(\pi) = [1, 1] \wedge [1, 1] \wedge [1, 0] = [1, 0]$$

This reveals that while the USER role is granted access at e_0, it will ultimately fail to execute the complete chain due to restrictions enforced at e_2. This matrix logic is visually summarized in Table III.

If a role is permitted at the system's entry ($\mathbf{a}(e_0)[r] = 1$) but blocked further down the chain, generating a $1 \to 0$ vector transition, the framework isolates it as a strong candidate for an authorization anomaly.

TABLE III
CHAIN-AWARE AUTHORIZATION EVALUATION FOR $\pi = \langle e_0, e_1, e_2 \rangle$, HIGHLIGHTING A DOWNSTREAM POLICY INCONSISTENCY FOR THE USER ROLE AT e_2.

Role	$\mathbf{a}(e_0)$	$\mathbf{a}(e_1)$	$\mathbf{a}(e_2)$
ADMIN	1	1	1
USER	1	1	0
Chain-aware	$\mathbf{a}^{chain}(\pi) = [1, 0]$		

D. Scenario Generation with Expected Outcomes

Moving beyond stochastic fuzzing techniques [20], our framework relies on a deterministic scenario generator that translates the policy-enriched IR into concrete test scenarios. By methodically traversing the ICFG, the system maps out distinct authorization pathways and mathematically calculates the exact expected outcome (the Oracle) for every role.

For each system endpoint, the generator reconstructs its complete upstream execution lineage, defined as $C = \{n_0, n_1, \ldots, n_k\}$. To formulate the Oracle, it derives the **Effec-**

tive AuthZ Vector (V_{eff}) by intersecting the permitted roles across the entire trace. A specific role r is deemed legitimate for the flow only if it fulfills the permission predicates at every node:

$$V_{eff}(r) = \bigwedge_{i=0}^{k} \text{Allow}(n_i, r) \tag{2}$$

If $r \in V_{eff}$, the framework dictates a `2xx` (Authorized) expected response; otherwise, it anticipates a `403 Forbidden`. This mechanism guarantees a ground-truth oracle even across convoluted, multi-hop transactions where middle-tier services might silently reject requests. Prior to actual test generation, the analyzer contrasts upstream (n_i) and downstream (n_{i+1}) permissions to categorize latent architectural defects:

- **Over-Permissive Downstream:** Identified when an upstream node n_i heavily restricts access (e.g., `ADMIN_ONLY`), but the downstream node n_{i+1} permits broader access. If the upstream gateway is bypassed, this configuration risks horizontal privilege escalation.
- **Restrictive Upstream:** Identified when the initial entry point is vastly more restrictive than the downstream services it invokes. While technically secure, this frequently points to logic flaws or "dead code" where downstream features are impossible for certain user roles to reach.
- **Policy Exposure:** Utilizing a heuristic feature extractor (which scans for contextual keywords such as *'payment'*, *'ssn'*, or *'profile'*), the framework evaluates policy sensitivity. If an endpoint flagged as `PII` or `FINANCIAL` is mapped as `PUBLIC` (Vector -1), it is immediately highlighted as a critical *Policy Exposure* anomaly. Listing 1 demonstrates a JSON scenario pinpointing this exact pattern within the Train-Ticket benchmark.

E. Prompt Construction and LLM-Constrained Test Generation

To bridge theoretical modeling and runtime execution, we deploy GPT-5.1 to translate these structural scenarios into executable HTTP tests. The generated scenarios dictate **what to test**, while a library of reusable prompt templates dictates **how to execute**. To maximize few-shot reasoning performance, scenarios are categorized into three operational modes:

1) **Entry-Point Scenarios ($d = 0$):** Centered on isolated, local matrix validation. These trigger a *Low Data Awareness* flag, allowing the LLM to populate requests with synthetic, superficial data since the goal is merely to confirm basic `ALLOW/DENY` routing.
2) **Consistency Scenarios ($d > 0$):** Geared toward end-to-end traversal. These trigger a *High Data Awareness* flag, mandating that the LLM synthesize semantically coherent payloads (such as valid foreign keys) to ensure the request can deeply penetrate the call graph without aborting via a `500 Internal Error`.
3) **Chain-of-Thought Triggers:** Activated upon detecting specific architectural conflicts (such as `RestrictiveUpstream`). The framework injects a Chain-of-Thought (CoT) prompt template. As detailed in Listing 1, this strictly forces the LLM to articulate the reasoning behind the conflict prior to generating the test assertion.

```
{
  "scenario_id": "scn_userservice:88dc3f08
    ...",
  "type": "DownstreamPolicyConsistency",
  "endpoint": "/api/v1/userservice/users/
    register",
  "expected_status_by_role": {
    "ROLE_ADMIN": "2xx",
    "ROLE_USER": "2xx"
  },
  "chain_permissions": {
    "userservice": { "allowed_roles": ["
    ROLE_ADMIN", "ROLE_USER"] },
    "authservice": { "allowed_roles": ["
    ROLE_ADMIN", "ROLE_USER"] }
  },
  "policy_inconsistencies": [
    {
      "role": "PUBLIC",
      "type": "PolicyExposure",
      "details": { "reason": "Sensitive PII
    endpoint marked PUBLIC" }
    }
  ],
  "llm_prompt_type": "CHAIN_OF_THOUGHT",
  "sensitivity_type": "PII"
}
```

Listing 1. Fragment of a generated testing scenario for a Train-Ticket endpoint, highlighting a detected downstream policy exposure anomaly.

Comprehensive versions of all reusable prompt templates utilized across these scenarios are published in our open-source Zenodo repository [28].

These prompt architectures enforce rigid syntactic rules, mandating specific token placeholder formats (e.g., {{`TOKEN_ROLE_X`}}), standardizing test categories (allowed, denied, anonymous, corrupted token), and enforcing rigid expected outcomes (`allowed` $\rightarrow$ `2xx`, `denied` $\rightarrow$ `403`, `anonymous/invalid` $\rightarrow$ `401`).

For every evaluated scenario, the user prompt is compiled using two distinct components: (1) an `INPUT DATA` JSON block supplying endpoint metadata, body schemas, parameter constraints, and role expectations, and (2) a directive `TASK` block that commands the LLM to output exactly one test case per role constraint. Crucially, for downstream scenarios, the `INPUT DATA` is enriched with chain context, including downstream permissions and flagged inconsistencies, ensuring the LLM evaluates the full chain's authorization model rather than just the isolated entry point.

F. Test Execution Harness and Validation

In the final phase, the generated test specifications are executed against a live, deployed microservice environment. The custom testing harness systematically handles authentication provisioning, constructs the HTTP requests, executes them with strict timeouts, and captures the responses.

Token provisioning. The harness dynamically resolves the required authentication state for each specific test case. Anonymous interactions simply drop the `Authorization` header. Role-specific test cases utilize mapped credentials to fetch a fresh JSON Web Token (JWT) employing bounded exponential backoff to smooth over transient network hiccups, which is then injected via `Authorization: Bearer <token>`. To test authentication failures, invalid-token scenarios deterministically mutate a valid JWT (such as altering its cryptographic signature).

Platform-independent execution. By adopting a lightweight, `curl`-based execution harness, we intentionally decouple our validation phase from language-heavy test frameworks. Research indicates that LLM-generated tests targeting environments like JUnit frequently fail due to hallucinatory dependencies or complex setup states [23]. Transpiling the tests into raw HTTP transactions bypasses these compilation hurdles, resulting in highly reproducible, fast, and easily inspectable execution commands.

Execution and scalability. Materializing the requests involves concatenating the base URL, endpoints, headers, query parameters, and JSON payloads. This stateless `curl` abstraction allows our runner to achieve massive throughput through process-level parallelization and request batching, efficiently capturing full HTTP responses (status, headers, and body) into a structured log format.

Validation. The final validation step cross-references the actual HTTP status codes against the LLM-derived Oracle, optionally evaluating specific body predicates for `2xx` responses. Invalid token scenarios are strictly required to fail (yielding a 401 or 403). Notably, any downstream 5xx server errors are strictly categorized as dependency failures, preventing them from being mistakenly logged as successful authorization blocks. Comprehensive metadata regarding test outcomes and any observed deviations is persisted for final aggregation.

IV. Results

In this section, we provide an empirical evaluation of our methodology against existing state-of-the-art benchmarks. To support reproducibility and further research, our implementation, generated scenarios, prompt templates, and raw LLM responses are available in our open-source repository [28].

A. Experimental Setup

To rigorously assess downstream authorization (AuthZ) testing, we utilized the 20-service configuration of *Train-Ticket*, which serves as a gold standard for complex microservice benchmarking [29]. We evaluated the system across three distinct AuthZ policy states: (1) a baseline uncorrupted state, (2) a moderate drift state with 60% random policy modifications, and (3) a severe drift state with 80% modifications. These configurations establish a robust ground truth for evaluating drift detection capabilities. While Train-Ticket encompasses 259 distributed execution paths, our analysis focused on the 23.5% of paths that participate in deep, multi-hop service interactions where AuthZ inconsistencies typically manifest. The maximum observed path depth reached 17 remote calls, ensuring that our evaluation targets deeply nested logic rather than trivial entry-point routing.

Our selection process yielded 13 representative endpoints spanning all observed graph depths and HTTP methods. The static analysis component utilized *JavaParser* for metadata extraction, while the Endpoint Behavior Graph (EBG) and AuthZ matrices were computed using a Python-based analyzer. Test generation was performed using the `GPT-5.1 Codex Mini` model in a few-shot configuration. We applied a temperature of 0.4 and a 4,000-token context window to balance schema adherence with the variance required for realistic mock data generation. We compared our approach against three baseline paradigms: EvoSuite, EvoMaster (v3.4.0), Augmented EvoMaster (AEM v5.1.0), and a state-of-the-art Formal Methods (FM) drift analysis framework [13].

B. Static Analysis and EBG Construction

The efficacy of our methodology depends heavily on the accuracy of the EBG construction. Our static analyzer demonstrated high precision, identifying 418 remote call sites within Train-Ticket and successfully resolving 399 of them. This represents a 95.5% resolution rate. Manual inspection of the remaining 19 unresolved sites revealed that they relied on dynamic URI construction, such as paths generated within runtime loops or those dependent on complex database queries. To maintain the integrity of the security analysis, these unresolved nodes were maintained as unlinked endpoints to prevent the hallucination of false policy mappings. This minimal unresolved rate did not significantly impede the framework's ability to detect downstream drift, as confirmed by the subsequent scenario coverage.

C. Validity of the Scenario Generator (RQ1)

The Scenario Generator serves as the critical bridge between theoretical policy models and executable tests. Because our methodology utilizes a policy-enriched Intermediate Representation (IR) as its single source of truth, the generator achieved 100% AuthZ scenario coverage across all baseline and drift-injected branches. As summarized in Table IV, the generator maintained high levels of semantic correctness. Unlike AEM and EvoMaster, which frequently produced tests triggering generic 500-level system errors, our generator accurately mapped roles to expected outcomes and parameter specifications.

TABLE IV
Quality evaluation of the test scenario generator for Train-Ticket.

Metric Category	Performance
Role and Path Coverage	100%
Correct Expected Outcomes	93.32%
Correct Role Assignment	82.05%
Correct Parameter Specification	100%
True Positive / True Negative Rate	100% / 100%
Precision / Recall / F1 Score	100% / 100% / 100%

A defining characteristic of our approach is its adherence to Explainable AI principles. Traditional SBST tools often produce opaque assertions that lack security context. In contrast, our framework requires the LLM to articulate a causal rationale linking the binary AuthZ vector in the IR to the expected HTTP status code. Throughout our evaluation, the model produced 100% accurate natural language justifications, identifying specific anomalies such as *OverPermissiveDownstream* mismatches. This transforms the test oracle from a simple status check into a verifiable security claim.

D. Test Quality and Execution Capability (RQ2)

Table V highlights the empirical performance disparity between our semantic generation approach and traditional SBST tools. For each distributed path, our framework generated a suite of four AuthZ-specific calls, totaling 52 tests per policy state. The data reveals a significant "semantic gap" in existing tools; while EvoMaster is syntactically accurate, it achieved less than 50% alignment with actual downstream scenarios. EvoSuite failed to generate any semantically meaningful endpoint tests for these distributed services.

TABLE V
COMPARATIVE ANALYSIS OF TEST GENERATION QUALITY ACROSS DIFFERENT METHODOLOGIES. VALUES REPRESENT THE AVERAGE PERCENTAGE (%) OF GENERATED TESTS.

Metric	Ours	AEM	EvoMaster	EvoSuite
Number of Tests	52	52	54	4
Correct Endpoints	100.0	92.3	100.0	0.0
Expected Status Codes	100.0	92.3	100.0	0.0
Inline w/ Downstream Scenarios	100.0	15.4	46.1	0.0
Meaningful Assertions	100.0	53.9	92.3	0.0
Explicitly Verifies AuthZ	**100.0**	**76.9**	**84.6**	**0.0**
Tests Boundary Conditions	100.0	23.8	15.4	0.0
Runs Without Error	97.4	92.3	100.0	0.0
Throws Exceptions/Anomalies	2.6	7.7	0.0	100.0

Our framework achieved 100% semantic validity, correctly targeting endpoints and verifying AuthZ invariants. In terms of execution reliability, our platform-independent HTTP specifications reached a 97.4% error-free execution rate. This avoids the heavy instrumentation requirements of tools like EvoMaster, which are often tightly coupled to specific runtime agents. On average, our pipeline required 2.82 seconds per scenario for static analysis and 13.52 seconds for remote LLM inference, representing a manageable overhead for CI/CD integration.

E. Drift Detection Accuracy

We further evaluated how effectively the generated tests identify security regressions compared to a Formal Methods (FM) baseline. We defined the positive class as authorization consistency and the negative class as authorization drift. As shown in Table VI, the FM approach struggled significantly with business-logic policy violations, failing to identify true drifts and misclassifying multiple drifting cases as secure. This high False Positive Rate (FPR) suggests that FM under-reports critical security risks in complex microservice environments.

TABLE VI
CONFUSION MATRIX AND PERFORMANCE METRICS FOR DRIFT DETECTION IN TRAIN-TICKET.

Approach	TP	TN	FP	FN	Acc.	Prec.	Rec.	F1
FM [13]	18	0	16	5	0.462	0.529	0.783	0.631
Ours	**22**	**16**	**0**	**1**	**0.963**	**1.000**	**0.957**	**0.978**

In contrast, our methodology achieved a zero False Positive Rate, ensuring that no actual authorization drifts were missed. By identifying 16 true drifts with only one false alarm, our framework attained an F1-score of 0.978. These results suggest that downstream policy awareness constitutes a fundamental capability boundary that current search-based tools cannot cross without structural guidance from a policy-enriched IR.

F. Threats to Validity

Several factors may influence the validity of our findings. Regarding construct validity, our methodology focuses on backend routing and excludes User Interface testing. To ensure our results mirror realistic security threats, we mapped all injected drifts to established MITRE CWE patterns, specifically CWE-285 and CWE-639. Internal validity was maintained by separating the authorship into independent groups for data extraction and validation. While we utilized GPT-5.1 in a few-shot configuration without fine-tuning, this represents a conservative performance baseline and prevents the threat of model over-fitting.

External validity threats include our focus on the Java Spring Cloud ecosystem. While our static parser is language-specific, the underlying concept of a policy-enriched IR is transferrable to any component-based architecture. However, our reliance on static analysis assumes a relatively low percentage of dynamically resolved calls. While this holds for systems using declarative clients, it may be challenged in architectures relying on database-driven routing or heavy reflection. Finally, to mitigate conclusion threats stemming from the lack of a standardized authorization drift dataset, we utilized the established Train-Ticket benchmark and conducted manual inspections to verify all detected vulnerabilities.

V. LESSONS LEARNED, IMPLICATIONS, AND OPEN CHALLENGES

The application of LLMs to distributed authorization testing yields several critical insights into the intersection of generative AI and formal security verification. Primarily, models such as GPT-5 demonstrate a strong capacity to translate policy-aware scenarios into semantically aligned API tests, provided they are strictly grounded in structured inputs like endpoint schemas, role contexts, and deterministic authorization outcomes. While the model reliably captures intended security semantics, it can occasionally struggle with structural strictness, producing malformed payloads or incorrect HTTP semantics. To ensure continuous executability, our pipeline utilizes an automated, schema-driven sanitization layer that repairs model outputs without manual intervention. Although

this guarantees syntactical correctness, it highlights a persistent gap between structural validity and deep authorization reachability, as synthesizing complex, domain-valid application state remains a hurdle for pure LLM generation.

Furthermore, shifting the testing paradigm from isolated entry points to comprehensive downstream call chains uncovers vulnerabilities that remain entirely invisible to traditional methods. By evaluating authorization feasibility across the entire service continuum, our framework successfully detected a latent defect representative of a broader class of propagation errors: a downstream service failing to forward the requisite authorization header, thereby causing legitimate, authenticated upstream requests to fail. This underscores that deep authorization validation inevitably converges with integration testing. When downstream endpoints fail, it is frequently due to missing domain state rather than pure authorization logic. Consequently, robust downstream testing should include controlled data provisioning or execution-flow validation to ensure authorization points are properly exercised and not bypassed by superficial data validation issues.

A. Implications for Practice and Research

From an operational perspective, this framework functions as an automated security gate within CI/CD pipelines, enabling the continuous, early detection of authorization drift prior to deployment. By automating the validation of cross-service security behaviors, organizations can reduce their reliance on periodic manual audits and confidently accelerate the evolution of complex microservice ecosystems.

Scientifically, this work provides strong empirical evidence that grounding LLM reasoning in a deterministic, static intermediate representation resolves the unreliability typically associated with generative testing. It advocates for a paradigm shift toward structure-aware, AI-assisted testing pipelines. Rather than utilizing LLMs as unconstrained, black-box generators, they are most effective when operating as reasoning engines within analytically constrained environments.

B. Open Challenges

Despite these advancements, several structural and operational challenges remain unresolved. Chain-aware authorization testing naturally introduces combinatorial state explosion across roles, endpoints, and call paths. Addressing this requires the development of incremental, change-impact-driven testing strategies to prioritize critical paths and maintain computational tractability. Additionally, while static analysis effectively captures application-layer logic, authorization rules enforced at the infrastructure level, such as within API gateways, service meshes, or external middleware, may elude static extraction, thereby limiting the completeness of the generated testing oracle. Generating minimal system state and semantically valid request data also remains essential for penetrating deep authorization checks and avoiding false negatives.

Finally, widespread industrial adoption is currently hindered by a lack of organizational trust in probabilistic assertions [30]. Integrating generative AI tools into rigorous CI/CD environments demands that generated tests be entirely deterministic and free of flakiness. Furthermore, the inference latency and financial cost of querying massive foundation models for every build are prohibitive at scale. Overcoming this barrier requires distilling these capabilities into smaller, task-specific fine-tuned models. These models should preserve strong reasoning quality while significantly improving speed and reducing resource usage.

VI. Conclusion

This work addresses a critical vulnerability in modern distributed architectures: the "authorization blindspot", where access control drift occurs deep within downstream service dependencies, invisible to perimeter gateways and traditional testing. To overcome this limitation, we introduced the first fully automated framework that systematically bridges formal static analysis with Generative AI. By constructing a policy-enriched IR and mapping cross-service communication via EBG, our approach establishes a deterministic foundation. This grounds an LLM (GPT-5.1) to synthesize executable, downstream-aware HTTP test suites that rigorously validate authorization intent across entire execution chains.

Our empirical evaluation on the Train-Ticket benchmark demonstrates a decisive advantage over state-of-the-art baselines. While SBST tools such as EvoMaster and EvoSuite struggled to navigate deep authorization logic, verifying fewer than 46% of downstream cases and producing invalid or hollow assertions, our framework achieved 100% semantic validity and verified authorization in 97.4% of tests. Furthermore, when compared against state-of-the-art Formal Methods for drift detection, our methodology detected all injected authorization policy violations while maintaining a flawless 0% False Positive Rate (achieving an F1-score of 0.978). This eliminates the noise that plagues static security analysis.

Broadly, this research establishes a new paradigm for AI-assisted security verification. We provide empirical evidence that Large Language Models, when strictly grounded in formal architectural structure, can transcend their roles as stochastic text generators to function as rigorous, explainable verification engines. By transforming opaque policy configurations into explicitly testable, chain-aware assertions, this framework advances both the theory of distributed verification and the practical reality of secure CI/CD pipelines. Ultimately, this methodology lays the groundwork for intelligent, self-verifying cloud architectures capable of continuously monitoring, explaining, and remediating security drift in real-time.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 2409933.

References

[1] F. Dai, M. A. Hossain, and Y. Wang, "State of the art in parallel and distributed systems: Emerging trends and challenges," *Electronics*, vol. 14, no. 4, p. 677, 2025. [Online]. Available: https://www.mdpi.com/2079-9292/14/4/677

[2] T. Cerny, G. Goulis, and A. S. Abdelfattah, "Towards change impact analysis in microservices-based system evolution," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2025, pp. 159–169. [Online]. Available: https://ieeexplore.ieee.org/document/10992369/

[3] B. Smith. (2018) Project strobe: Protecting your data, improving our third-party APIs, and sunsetting consumer google+. [Online]. Available: https://blog.google/technology/safety-security/project-strobe/

[4] B. Chappell. Uber pays $148 million over yearlong cover-up of data breach. [Online]. Available: https://www.npr.org/2018/09/27/652119109/uber-pays-148-million-over-year-long-cover-up-of-data-breach

[5] OWASP. A01:2021 – broken access control. [Online]. Available: https://owasp.org/Top10/2021/A01_2021-Broken_Access_Control/

[6] S. Vogl, S. Schweikl, G. Fraser, A. Arcuri, J. Campos, and A. Panichella, "Evosuite at the sbst 2021 tool competition," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2021, pp. 28–29.

[7] A. Arcuri, "EvoMaster: Evolutionary multi-context automated system test generation," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 394–397. [Online]. Available: https://ieeexplore.ieee.org/document/8367066/

[8] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 748–758. [Online]. Available: https://ieeexplore.ieee.org/document/8811961/

[9] E. Viglianisi, M. Dallago, and M. Ceccato, "RESTTESTGEN: Automated black-box testing of RESTful APIs," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 142–152. [Online]. Available: https://ieeexplore.ieee.org/document/9159077/

[10] M. Kim, S. Sinha, and A. Orso, "LlamaRestTest: Effective REST API testing with small language models," *Proceedings of the ACM on Software Engineering*, vol. 2, pp. 465–488, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3715737

[11] C. Cao, A. Panichella, and S. Verwer, "Automated test-case generation for rest apis using model inference search heuristic," in *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2025, pp. 29–40.

[12] A. Salahirad, G. Gay, and E. Mohammadi, "Mapping the structure and evolution of software testing research over the past three decades," *Journal of Systems and Software*, vol. 195, p. 111518, 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121222001947

[13] C. Wojtak, "Formal methods for verifying authorization policy in microservice systems," in *2025 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2025, pp. 241–242. [Online]. Available: https://ieeexplore.ieee.org/document/11126128/

[14] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10440574/

[15] M. A. Uddin, "Graph-based LLM prompting for scalable microservice API testing," in *2025 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2025, pp. 243–244. [Online]. Available: https://ieeexplore.ieee.org/document/11126137/

[16] T. Miao, A. I. Shaafi, and E. Song, "Systematic mapping study of test generation for microservices: Approaches, challenges, and impact on system quality," *Electronics*, vol. 14, no. 7, p. 1397, 2025. [Online]. Available: https://www.mdpi.com/2079-9292/14/7/1397

[17] C. Augusto, J. Morán, A. Bertolino, C. De La Riva, and J. Tuya, "Software system testing assisted by large language models: An exploratory study," in *Testing Software and Systems*, H. D. Menéndez, G. Bello-Orgaz, P. Barnard, J. R. Bautista, A. Farahi, S. Dash, D. Han, S. Fortz, and V. Rodriguez-Fernandez, Eds. Springer Nature Switzerland, 2025, vol. 15383, pp. 239–255. [Online]. Available: https://link.springer.com/10.1007/978-3-031-80889-0_17

[18] D. Das, A. Walker, V. Bushong, J. Svacina, T. Cerny, and V. Matyas, "On automated RBAC assessment by constructing a centralized perspective for microservice mesh," *PeerJ Computer Science*, vol. 7, p. e376, 2021. [Online]. Available: https://peerj.com/articles/cs-376

[19] B. Uzun and B. Tekinerdogan, "Detecting deviations in the code using architecture view-based drift analysis," *Computer Standards & Interfaces*, vol. 87, p. 103774, 2024. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0920548923000557

[20] J. Cheng, D. Fang, Z. Gu, S. Lv, S. Si, and L. Sun, "Discovering PLC web application vulnerabilities impacting physical control using LLM-based fuzzing," in *Wireless Artificial Intelligent Computing Systems and Applications*, Z. Cai, Y. Zhu, Y. Wang, and M. Qiu, Eds. Springer Nature Singapore, 2025, vol. 15686, pp. 49–61. [Online]. Available: https://link.springer.com/10.1007/978-981-96-8725-1_5

[21] X. Liu, X. Sun, L. Bo, Y. Hu, X. Liu, and Z. Ye, "Evaluating the test adequacy of benchmarks for LLMs on code generation," *Journal of Software: Evolution and Process*, vol. 37, no. 7, p. e70034, 2025. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/smr.70034

[22] E. M. Pasca, R. Erdei, D. Delinschi, and O. Matei, "Enhancing API security testing against BOLA and authentication vulnerabilities through an LLM-enhanced framework," in *The 19th International Conference on Soft Computing Models in Industrial and Environmental Applications SOCO 2024*, H. Quintian, E. Corchado, A. Troncoso Lora, H. Perez Garcia, E. Jove, J. L. Calvo Rolle, F. J. Martinez De Pison, P. Garcia Bringas, F. Martinez Alvarez, A. Herrero Cosio, and P. Fosci, Eds. Springer Nature Switzerland, 2025, vol. 889, pp. 231–240. [Online]. Available: https://link.springer.com/10.1007/978-3-031-75010-6_23

[23] ——, "Augmenting api security testing with automated llm-driven test generation," in *International Joint Conferences*, H. Quintián, E. Corchado, A. Troncoso Lora, H. Pérez García, E. Jove, J. L. Calvo Rolle, F. J. Martínez de Pisón, P. García Bringas, F. Martínez Álvarez, Á. Herrero Cosío, and P. Fosci, Eds. Cham: Springer Nature Switzerland, 2024, pp. 112–121.

[24] S. Fischer and C. Klammer, "Automating invariant filtering: Leveraging LLMs to streamline test oracle generation," in *Balancing Software Innovation and Regulatory Compliance*, J. Fischbach, R. Ramler, D. Winkler, and J. Bergsmann, Eds. Springer Nature Switzerland, 2025, vol. 544, pp. 51–71. [Online]. Available: https://link.springer.com/10.1007/978-3-031-89277-6_4

[25] R. Nooyens, T. Bardakci, M. Beyazıt, and S. Demeyer, "Test amplification for REST APIs via single and multi-agent LLM systems," in *Testing Software and Systems*, S. Bonfanti and G. A. Papadopoulos, Eds. Springer Nature Switzerland, 2025, vol. 16107, pp. 161–177. [Online]. Available: https://link.springer.com/10.1007/978-3-032-05188-2_11

[26] S. Y. Shin, F. Pastore, D. Bianculli, and A. Baicoianu, "Towards generating executable metamorphic relations using large language models," in *Quality of Information and Communications Technology*, A. Bertolino, J. Pascoal Faria, P. Lago, and L. Semini, Eds. Springer Nature Switzerland, 2024, vol. 2178, pp. 126–141, series Title: Communications in Computer and Information Science. [Online]. Available: https://link.springer.com/10.1007/978-3-031-70245-7_9

[27] A. S. Abdelfattah, T. Cerny, J. Yero Salazar, X. Li, D. Taibi, and E. Song, "Assessing evolution of microservices using static analysis," *Applied Sciences*, vol. 14, no. 22, p. 10725, 2024. [Online]. Available: https://www.mdpi.com/2076-3417/14/22/10725

[28] "A preliminary study on the quality of LLM-generated authorization tests for microservices," 2026. [Online]. Available: https://doi.org/10.5281/zenodo.19230026

[29] S. Smith, E. Robinson, T. Frederiksen, T. Stevens, T. Cerny, M. Bures, and D. Taibi, "Benchmarks for end-to-end microservices testing," in *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2023, pp. 60–66. [Online]. Available: https://ieeexplore.ieee.org/document/10254752/

[30] C. Augusto, A. Bertolino, G. D. Angelis, F. Lonetti, and J. Morán, "Large language models for software testing: A research roadmap," 2025. [Online]. Available: http://arxiv.org/abs/2509.25043