SAINT: Service-level Integration Test Generation with Program Analysis and LLM-based Agents

Rangeet Pan
rangeet.pan@ibm.com
IBM Research
Yorktown Heights, NY, USA

Raju Pavuluri
pavuluri@us.ibm.com
IBM Research
Yorktown Heights, NY, USA

Ruikai Huang
rkh@gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Rahul Krishna
rkrsn@ibm.com
IBM Research
Yorktown Heights, NY, USA

Tyler Stennett*
tstennett3@gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Alessandro Orso
alessandro.orso@cc.gatech.edu
University of Georgia
Athens, GA, USA

Saurabh Sinha
sinhas@us.ibm.com
IBM Research
Yorktown Heights, NY, USA

Abstract

Enterprise applications are typically tested at multiple levels, with service-level testing playing an important role in validating application functionality. Existing service-level testing tools, especially for RESTful APIs, often employ fuzzing and/or depend on OpenAPI specifications which are not readily available in real-world enterprise codebases. Moreover, these tools are limited in their ability to generate functional tests that effectively exercise meaningful scenarios. In this work, we present SAINT, a novel white-box testing approach for service-level testing of enterprise Java applications. SAINT combines static analysis, large language models (LLMs), and LLM-based agents to automatically generate endpoint and scenario-based tests. The approach builds two key models: an endpoint model, capturing syntactic and semantic information about service endpoints, and an operation dependency graph, capturing inter-endpoint ordering constraints. SAINT then employs LLM-based agents to generate tests. Endpoint-focused tests aim to maximize code and database interaction coverage. Scenario-based tests are synthesized by extracting application use cases from code and refining them into executable tests via planning, action, and reflection phases of the agentic loop. We evaluated SAINT on eight Java applications, including a proprietary enterprise application. Our results illustrate the effectiveness of SAINT in coverage, fault detection, and scenario generation. Moreover, a developer survey provides strong endorsement of the scenario-based tests generated by SAINT. Overall, our work shows that combining static analysis with agentic LLM workflows enables more effective, functional, and developer-aligned service-level test generation.

*Author was an intern at IBM Research at the time of this work.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773185

ACM Reference Format:
Rangeet Pan, Raju Pavuluri, Ruikai Huang, Rahul Krishna, Tyler Stennett, Alessandro Orso, and Saurabh Sinha. 2026. SAINT: Service-level Integration Test Generation with Program Analysis and LLM-based Agents. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773185

1 Introduction

Enterprise applications are large, multi-tiered systems with complex business logic. To gain confidence in their correctness, such applications are tested at multiple levels, each focusing on different validation goals. Unit testing checks individual implementation units (e.g., methods, functions, or classes) in isolation, whereas end-to-end testing exercises functional flows across application tiers. Between these levels, service-level testing works at the service layer of the application, with the goal of validating the service endpoints and server-side logic. It is typically guided by coverage goals over service operations, their parameters, and reachable code, as well as operation sequences that reflect functional flows or use cases.

In this work, we focus on improving service-level (or API-level) testing of enterprise Java applications. Our goal is two-fold: to generate high-coverage tests for individual service operations or endpoints, and to create scenario-based tests that exercise sequences of operations corresponding to coherent use cases. Although many service-level testing techniques exist—targeting RESTful APIs [3, 4, 9, 20–22, 24, 28, 29, 32, 33, 42, 52, 54], GraphQL APIs (e.g., [7, 19]) and the older WSDL-based services (e.g. [6])—they have key limitations that restrict their applicability for functional test generation on enterprise Java applications. First, most REST API testing tools function as fuzzers and do not produce test cases. Second, techniques that generate tests focus on maximizing code coverage [3] or infer sequences from producer-consumer or resource-based dependencies [42]. However, meaningful operation sequences can exist without such dependencies. Finally, most approaches rely on formal service specifications—typically OpenAPI [36]—which are often not available for enterprise applications.

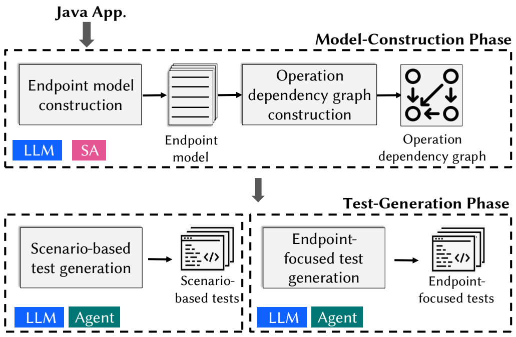

Figure 1: Overview of our approach.

We present a new white-box technique, called SAINT, for service-level testing that combines static program analysis with the power of large language models (LLMs)—leveraging their planning, reasoning, and reflection capabilities through agentic workflows to enhance test generation effectiveness. Our approach (shown in Figure 1) generates *endpoint-focused tests* that exercise each service endpoint with the goal of maximizing code coverage, and *scenario-based tests* that focus on covering meaningful use cases of the application for functional testing. The approach consists of two phases: model-construction and test-generation.

The model-construction phase analyzes the application under test to identify service endpoints of the application and constructs an *endpoint model* that incorporates (1) syntactic information for an endpoint that consists of endpoint path, parameter names, and parameter types, and (2) semantic information for an endpoint with inter-parameter dependencies [31] and parameter value constraints. Additionally, we construct an *operation dependency graph* (ODG) that captures ordering constraints between endpoints resulting from different types of dependencies. Both these models are constructed using a combination of static analysis and carefully crafted LLM prompts, with suitable in-context learning examples.

The test-generation phase of SAINT uses the endpoint model and the ODG to create two test types based on user intent: endpoint-focused tests and scenario-based tests. The endpoint-focused tests explore each service endpoint with varied inputs to maximize code coverage, with emphasis on covering database interaction points in code. To create these tests, SAINT constructs an LLM prompt for each endpoint using syntactic, semantic, and ordering constraints from the ODG, sends it to the LLM, and extracts parameter value sets from the received LLM response. It then builds concrete HTTP requests and executes them against the deployed APIs, while monitoring coverage of application code. SAINT incorporates a *repair agent* that attempts to fix invalid requests (i.e., requests for which the server returns 4xx response codes) and a *coverage-augmentation agent* that attempts to increase coverage of reachable code for an endpoint; both these agents implement an iterative planning, action execution (with an available set of tools), and reflection workflow.

To create scenario-based tests, which exercise meaningful application use cases, SAINT performs a sequence of LLM prompting to first extract test scenarios (in Gherkin-like syntax [12]) from the application code and then refine the scenarios into atomic blocks or test steps. These atomic blocks are input to a *test-generation agent* that attempts to reify a test scenario into an executable test case. This agent, like the repair and coverage-augmentation agents, implements a plan-act-reflect loop, with a suitable set of tools for executing actions. The output of the agent consists of Java test code fragments corresponding to the atomic blocks, which are then assembled, via another LLM call, into the final scenario-based test.

We evaluated SAINT on eight Java applications, including a proprietary enterprise application. Four of these applications have OpenAPI specifications; for these applications, we compared SAINT against EvoMaster, a state-of-the-art white-box test generation tool for RESTful APIs [3]. We assessed SAINT's effectiveness through coverage metrics and fault detection by server failure measurement. We evaluated the scenario-based tests for coverage, characteristics of the extracted scenarios, and developer perception, collected via a user survey. Finally, we analyzed SAINT's key components through an ablation study. Our results show that SAINT matches or considerably outperforms EvoMaster on code coverage achieved with the endpoint-focused tests, but is not as effective as EvoMaster in triggering server failures, with scope for improvement. In terms of scenario-based tests, SAINT effectively extracts scenarios that span multiple endpoints and are rated highly by developers, with more than 90% of the survey participants stating that they would test application scenarios similar to the extracted ones. The ablation study highlights the contributions of SAINT's key components.

The main contributions of the work are:

- A novel technique that combines static analysis with LLM prompting and agentic workflows to generate endpoint-focused and scenario-based service-level tests that aim to maximize coverage while also exercising meaningful use cases for functional testing.
- Empirical results showing SAINT's effectiveness in code coverage, fault detection, and scenario extraction, with developer feedback highlighting the value of SAINT-generated scenario-based tests.
- An artifact consisting of experiment scripts, data, and LLM prompts that is publicly available [49].

2 Motivation

In this section, we ground our discussion by presenting a reference example from Spring PetClinic [44], a multi-tier application that exposes multiple service endpoints requiring coordinated inputs, business rule enforcement across entities, and dynamic state-dependent database access. This example highlights some of the challenges for test generation. We then present an example test case generated by our tool, demonstrating how captures a realistic multi-endpoint sequence and validates non-trivial state-dependent behaviors.

Service-level testing challenges for a PetClinic endpoint. We consider the `.../ownerId/.../petId/visits/new` endpoint from PetClinic, shown in Figure 2. This example illustrates several challenges: 1 a value constraint on the `owner` parameter to ensure the provided `petId` belongs to the specified `ownerId`; 2 a temporal check to assert that the visit date is not in the past; 3 an implicit dependency that exists between the description text and the date (i.e., emergency visits must be within a day); 4 a condition that ties the surgery type to the date and the description. All the above clauses span entity relations, involving various inter-parameter dependencies, which cannot be adequately captured or represented using OpenAPI specifications or static analysis alone.

An illustrative test synthesized by our approach. The test shown in Figure 3 demonstrates how our approach addresses the

```java
@PostMapping("/owners/{ownerId}/pets/{petId}/visits/new")
public String processNewVisitForm(@ModelAttribute Owner owner,
    @ModelAttribute Pet pet, @Valid Visit visit, BindingResult result) {

    // Cross-path parameter relationship
    if (pet.getId() != owner.getId()) ........................... 1
        throw new IllegalArgumentException("Pet and owner mismatch");

    // Temporal constraint validation
    if (visit.getDate().isBefore(LocalDate.now())) ................. 2
        result.rejectValue("date", "past.date");

   // Context specific validation
   if (isEmergency(visit.getDescription()) ........................ 3
        if(visit.getDate().isAfter(LocalDate.now().addDays(1)))
            result.rejectValue("date", "not.same.day");

    // Multiparameter conditional validation
    if (isSurgeryVisit(visit.getDescription())) { .................. 4
        if (visit.getDate().isBefore(LocalDate.now().addDays(7)))
            result.rejectValue("date", "surgery.advance.booking");
        if (visit.getDescription.length() < 50)
            result.rejectValue("description", "detail.required");
    }
    if (result.hasErrors()) { return "pets/createOrUpdateVisitForm"; }
    return "redirect:/owners/{ownerId}"; }
```

Figure 2: Real-world endpoint from PetClinic with hard-to-test constraints

```java
@Test public void testCreateAndUpdatePetForAnOwner() {
    // Initialize creation form for a new pet
    Response response = given()
            .when().get("/owners/1/pets/new") ..................... 1
            .then().statusCode(200).extract().response();

    // Process creation form for a new pet for owner
    response = given()
            .contentType("application/x-www-form-urlencoded")
            .formParam("pet.name", "Buddy")
            .formParam("pet.birthDate", "2023-01-01")
            .when().post("/owners/1/pets/new") .................... 1
            .then().statusCode(200).extract().response();

    // Initialize update form for the pet of owner
    response = given().when()
            .get("/owners/1/pets/1/edit") ......................... 2
            .then().statusCode(200).extract().response();

    // Process update form to change date of birth
    response = given()
            .contentType("application/x-www-form-urlencoded")
            .formParam("pet.name", "Buddy") ....................... 3
            .formParam("pet.birthDate", "2022-01-01") ............. 3
            .when().post("/owners/1/pets/1/edit") ................. 2
            .then().statusCode(200).extract().response(); ......... 4}
```

Figure 3: A test synthesized by SAINT capturing these constraints in a realistic scenario.

Symbol	Type	Description
		API endpoint ($\mathcal{E}$) $\equiv$ (c, m, p, H, Π, I, D, R)
c	Σ^+	Fully qualified name of the class containing the endpoint method.
m	Σ^+	Signature of the API endpoint method
p	Σ^*	Endpoint path
H	$\mathbb{H}$	HTTP method. $\mathbb{H} = \{GET, POST, PUT, DELETE, PATCH\}$
Π	$\mathcal{P}^*$	List of endpoint parameters
I	$\mathcal{I}^*$	List of inter-parameter dependencies
D	$\mathcal{D}^*$	List of database operations
R	Σ^*	Response schema as dictionary or string reference
		Endpoint Parameter ($\mathcal{P}$) $\equiv$ (n, T, K, V, C, M, A)
n	Σ^*	Name of the parameter
T	Σ^*	Datatype of the parameter
K	$\mathcal{K}$	Parameter kind. $\mathcal{K} = \{query, path, body, header\}$
V	Σ^*	Value constraints (e.g., allowed strings, enum options)
M	Σ^*	Enclosing method for the parameter
C	Σ^*	Enclosing class for the parameter
A	Σ^*	List of annotations applied to the parameter
		Inter-parameter dependency ($\mathcal{I}$) $\equiv$ (R, Π, Γ)
R	$\mathbb{I}$	Inter-parameter dependency relation type.
Π	$\mathcal{P}^+$	List of involved parameters
Γ	Σ^*	Constraint logic associated with the relation
		Database operation ($\mathcal{D}$) $\equiv$ (F, C, L, M, O)
F	$\mathcal{F}$	Database framework. $\mathcal{F} = \{JDBC, JPA, JTA, ...\}$
C	Σ^*	Enclosing class name of the operation
M	Σ^*	Method signature where the DB access occurs
L	i	Line number where operation has been taken
O	$\mathcal{O}$	CRUD operation type. $\mathcal{O} = \{Create, Read, Update, Delete\}$

Figure 4: The endpoint model constructed by SAINT.

challenges in scenario-based testing. It constructs a coherent multi-endpoint test that maintains semantic consistency across endpoints with consistent use of owner ID 1 across the test (as seen in 1). It also enforces state-dependent constraints (as seen in 2, where the edit operations assume that the pet was successfully created beforehand), and embeds realistic validation logic (e.g., via valid form data entries on lines labeled 3). In summary, the example exposes cross-endpoint dependencies and produces a test case with valid inputs and assertions (e.g., final check on line 4), illustrating how our approach can perform effective service-level test generation.

3 Methodology

SAINT operates in two phases where Phase 1 constructs the endpoint model and the ODG and Phase 2 performs endpoint-based and scenario-based test generation.

3.1 Endpoint Model Construction

Figure 5 illustrates the construction our endpoint model using static analysis and LLM prompting. First, we identify endpoint methods in the application written in various Java frameworks. SAINT currently supports five Java frameworks: Jakarta [15], Spring [43], Struts [48], Stripes [47], and JDK HttpServer [17]. For Jakarta, Spring, and Struts, we rely on CLDK [8]. CLDK is a multi-language program analysis library that is built upon well-vetted static analysis tools, such as WALA [53], JavaParser [16], and Tree-sitter [51], and provides various analysis capabilities via Python APIs. We use these APIs to extract symbol table, call graph, and endpoint and database information and build custom analysis for this work. To identify endpoints, CLDK relies on a pattern-matching technique that was proposed in a previous work [1]. For Stripes and HttpServer, SAINT implements custom static analyses and LLM prompts to identify endpoints. These analyses leverage the symbol table information (e.g., method implementations, annotations, parameter details, etc.) and the call graph (constructed with Rapid Type Analysis [5]) obtained using the CLDK APIs. After identifying the endpoints, SAINT extracts detailed syntactic and semantic information for each endpoint to populate the endpoint model.

Figure 4 formally defines the endpoint model. An endpoint $\mathcal{E}$, corresponding to a method, is represented as an 8-tuple consisting of service class name c, method signature m, endpoint path p, HTTP method H, endpoint parameters Π, inter-parameter dependencies (IPDs) [31] $\mathcal{I}$, reachable database operations $\mathcal{D}$, and the response schema $\mathcal{R}$, representing the structure of the server response. The figure also shows the data type of each field in the model: a field is either a custom type defined within the model (e.g., $\mathbb{H}$ for HTTP methods), a string type (Σ), or an integer type (i).

Extracting endpoint path and HTTP method. Java frameworks use various patterns for declaring paths and operation types. Typically, these are class or method annotations that can be extracted via code parsing. However, in the case of HttpServer, a legacy framework that lacks annotation-based conventions, endpoint paths are specified in code, as shown in this example from LanguageTool [25]:

```java
void handleRequest(String path, ...) throws Exception {
    if (path.equals("languages")) {
      handleLanguagesRequest(httpExchange);
    } else if (path.equals("maxtextlength")) {...}
```

In this case, SAINT relies on LLMs to extract the paths. Although complex static analysis could, in principle, support applications built on HttpServer, we deemed such an effort unwarranted due to the framework's limited usage.

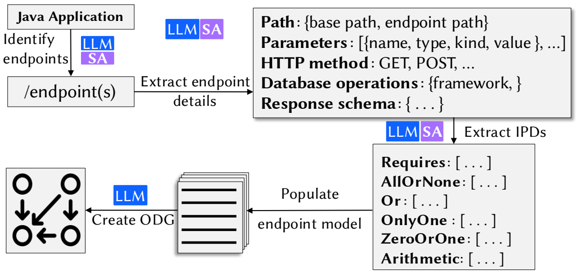

Figure 5: Construction of the endpoint model and ODG via static analysis (SA) and LLM prompting (LLM).

Extracting endpoint parameter details. The parameter information in the model (Figure 4) consists of the parameter name n, the parameter type T, the parameter kind K, value constraints V, enclosing method M and class C, and the associated annotations A.

Parameter names and types. In Spring and JAX-RS, parameters are declared in the endpoint method, allowing easy extraction of names and types (unless we encounter complex patterns `@ModelAttribute` in Spring). In other cases, such as Jakarta Servlets, more advance processing is required as shown in the below snippet. Here, the parameter `OrderProcessingMode` stored as a `String` and then converted to `int`. Here, `getParameter()` calls on `HttpRequest` are analyzed (see highlighted lines) where the parameter values are `String` typed (return type of `getParameter()`) and are subsequently typecast to `int`. These processes can happen anywhere in the call chain starting at the endpoint method.

```
void doConfigUpdate(HttpServletRequest req) throws Exception {
    String modeStr = req.getParameter("OrderProcessingMode");
    if (orderProcessingModeStr.isNotNull())
        int i = Integer.parseInt(modeStr);}// additional logic below
```

To handle such cases, SAINT performs a call-chain analysis, scoping it to the methods to which the `HttpRequest` object flows via parameter passing. Within this scope, it identifies parameter names and types via LLM prompting, including relevant code fragments from the call graph and a simple in-context example in the prompt.

The following example from JPetStore [18] (a Stripes application [47]) illustrates how parameters can be declared as class fields linked via getter-setter methods. The lines highlighted show how the `editAccount()` method passes an `account` object to `setAccount()`. As `AccountService` is annotated with `@Service`, its fields become parameters. In the `signon()` method, parameters are inferred from getters for `username` and `password`.

```
@Service public class AccountService {
    private Account account;
}
public class AccountActionBean {
    private Account account = new Account();
    public String getUsername() { return account.getUsername(); }
    public String getPassword() { return account.getPassword(); }
    public Resolution editAccount() {
    ↪   accountService.setAccount(account); }
    public Resolution signon() {
        account = accountService.getAccount(getUsername(),
        ↪   getPassword()); }}
```

Parameter kind. Parameter kind can be path, query, header, or body, indicating whether the parameter is included in the resource path, the request query string, or the request body. SAINT extracts this information using static analysis.

Parameter value constraints. Parameters can have value constraints enforced by code checks or annotations. In some frameworks, developers provide natural language examples to aid in generating corresponding values.

```
List<VariantAnnotation> getVariantAnnotation(@PathVariable
    @ApiParam(
        value = "Comma-separated variants (e.g., 1:g.123A>T,...)",
        required = true, allowMultiple = true)
    List<String> variants) { /*method body*/ }
```

In other cases, value constraints are specified in code, as seen in a DayTrader [10] fragment below: the `action` parameter supports specific values for request processing, while an invalid `action` value triggers a 4xx or 5xx response, depending on server settings.

```
public void task(HttpServletRequest req, ...)  {
    switch (req.getParameter("action")) {
    case 'q': {...} // quote
    case 'a': {...} // account
    case 'u': // update account profile
    // + 6 more cases ...}}
```

SAINT extracts the context for an endpoint parameter, including its annotations, type, and method body, and incorporates it into an LLM prompt to extract value constraints, instructing the LLM to produce the output in a structured format illustrated with an in-context example.

Enclosing method and class. These represent the method and class for parameter declaration. In most frameworks, they are the endpoint class and method. For Servlets, they indicate the method and its class from which a parameter is extracted from an `HttpServletRequest` instance, which could occur anywhere in the call chain starting at the endpoint method.

Extracting inter-parameter dependencies. API endpoints often have parameter dependencies that restrict valid request combinations. Prior work [31] identifies seven IPD types: `AllOrNone`, `Requires`, `OnlyOne`, `Or`, `ZeroOrOne`, `Arithmetic`, and `Complex`. For instance, the `OnlyOne` relation requires only one parameter, while the `AllOrNone` relation require all or none of the parameters to be present in a valid requst. To extract IPDs, SAINT prompts an LLM prompt with the relevant endpoint context, consisting of parameter names, types, and relevant method bodies. The prompt also includes IPD definitionsa and examples to teach the LLM about the relations and output formats. The LLM identifies the IPDs, determining the relation type R, the involved parameters Π, and the code constraints Γ, which are stored in the endpoint model (Figure 4).

Extracting database operation details. CLDK extracts database operations based on known APIs and supported database frameworks. It maps these APIs to their corresponding CRUD operations and records the location of each call in the analysis metadata. We leverage this information to identify lines of code that contain database interactions and prioritize their coverage while testing the individual endpoints, as such operations represent an important component of the functionality provided by the endpoints.

3.2 ODG Construction

The endpoint model captures syntactic and semantic details of each endpoint without considering inter-endpoint dependencies. For instance, in PetClinic [44], to add a pet, an owner ID must first be obtained via the endpoint listing all owners (`GET /owner`) or by adding a new owner (`POST /owner/{ownerid}`), demonstrating resource-based dependency. SAINT constructs the ODG to represent such dependencies. Moreover, SAINT creates a functional summary for each endpoint, used in Phase 2 to extract testing scenarios, and these summaries are linked to ODG nodes.

Formally, the ODG $G = (V, E)$ constitutes a directed graph, wherein the nodes are representative of endpoints (or service operations) and the edges delineate the dependencies among these endpoints. Each node $v \in V$ corresponding to an endpoint encompasses the model and a comprehensive functional summary of that endpoint. An edge $(v_1, v_2) \in E$ signifies the dependency of v_2 on v_1 through one of three distinct relational types: (1) *resource dependency*, where two endpoints are connected via path resources, exemplified by the PetClinic scenario; (2) *producer-consumer dependency*, involving an endpoint that produces a value and another endpoint that can take that value as input; and (3) *database dependency*, where one endpoint executes a write operation to a database and another endpoint retrieves data from the same database.

We use a RAFT-like approach [42] to analyze path parameters and HTTP methods (`GET`, `POST`, `DELETE`, etc.) for resource dependencies. For two other relations, we create prompts with endpoint and database details, and code fragments for LLMs to identify these relations. LLM calls also generate operation summaries based on provided endpoint code and other details.

3.3 Endpoint-focused Test Generation

Figure 6 shows the endpoint-focused test generation workflow, which includes: (1) generating and executing HTTP requests with parameters on services, (2) fixing invalid requests, (3) enhancing code coverage, and (4) converting requests into Java tests. All steps use LLMs, with steps 2 and 3 also using agentic workflows.

3.3.1 Generation and execution of HTTP requests. To generate endpoint parameter values, a prompt with endpoint model details (path, HTTP method, parameter names/types, constraints, and IPDs) and related code from the call graph is created. The LLM outputs parameter-to-value mappings, forming concrete HTTP requests. These requests are executed by SAINT against services monitored for coverage changes.

For each executed request, the technique checks the response code. For 4xx responses (indicating invalid requests), it invokes SAINT's repair agent to fix the request. For 2xx and 3xx responses, the technique invokes the coverage-augmentation agent to increase coverage of uncovered code reachable from the endpoint method. Both agents implement a plan-act-reflect loop consisting of a *planning step* where the LLM decides on the next course of action based on available information about the task at hand, an *action step* where the agent performs an action using an available set of tools, and a *reflection step* where the agent ranks the outcome of the action and sends feedback for the next iteration of the loop.

3.3.2 Tools for agents. We designed and implemented six tools that are suitable for the tasks to be performed by the repair and coverage agents (shown under "Available tools" in Figure 6).

(1) *Modify parameter value.* One of the common modifications needed to fix an invalid HTTP request or increase code coverage is adjusting parameter values. Selecting this action results in an LLM call with a prompt that includes relevant parameter details for the endpoint. When fixing HTTP requests, we also include the incorrect request along with an explanation generated by the LLM while selecting an action. Additionally, if the LLM determines that more code context is necessary, we include it in the prompt. For the coverage-augmentation, we supply the LLM with uncovered lines.

(2) *Modify parameter type.* This action is designed for handling application frameworks (e.g., Servlet) for which we obtain parameter types via LLM calls. If during the planning step, the agent reasons that the assigned type of a parameter causes an invalid request or uncovered lines, it can rectify that mistake via this action. The output of this action consists of new requests with parameter values generated in accordance with modified parameter types.

(3) *Modify IPD.* Similar to the parameter-type-update action, this action updates an IPD that was initially obtained via LLM prompting. Based on the new IPD, a new request is formed.

(4) *Update value constraint.* This action updates a previously extracted value constraint for a parameter, and forms a new HTTP request based on that.

(5) *Generate requests.* With this action, the agent generates more requests for the same endpoint or another endpoint, whose invocation may be a prerequisite for fixing an invalid request or covering more code in the endpoint under consideration.

(6) *Extract additional code context.* Often fixing an HTTP request or covering additional code requires code-related details that may not be available in the initial prompt. With this action, the agent can request more code context by selecting a CLDK API [8] to be invoked (from a list of APIs provided to the agent). For instance, the agent can request information about callees of an endpoint method.

3.3.3 Repair agent. The repair agent attempts to fix invalid HTTP requests. During the planning step, the agent is presented with the request details, along with LLM-generated problem summary of the error response from the server. As the raw server response can be overly verbose, making it hard to pinpoint the issue, we use an LLM to summarize the response, which makes the agent's planning step easier. Based on the presented information, the agent selects the next actions to execute (we limit this to two actions to control the computational cost). Along with the actions, the agent also generates the rationale for its decision. After executing a selected action, the agent reflects on the outcome by comparing the summarized server responses before and after the action to determine whether the action addresses the problem with the original invalid request. For instance, upon receiving a 4xx response and the corresponding server message, the repair agent selects suitable tools to regenerate the request and re-evaluates the response. This process continues until a 200 status code is obtained or a predefined upper bound on the number of attempts is reached. A scoring mechanism guides the agent's decisions at each iteration when the goal remains unmet.

3.3.4 Coverage-augmentation agent. This agent is tasked with generated HTTP requests targeted at covering specific (uncovered) lines of code. During the planning step, the agent is provided with the uncovered reachable lines of code and other relevant information about the tested endpoint. The agent selects up to two actions to execute next, along with the reasoning behind its choices. After performing the action, which results in execution of newly generated HTTP requests against the endpoint, the agent reflects on the outcome by comparing the previously uncovered lines with the newly covered lines after the execution of the new requests.

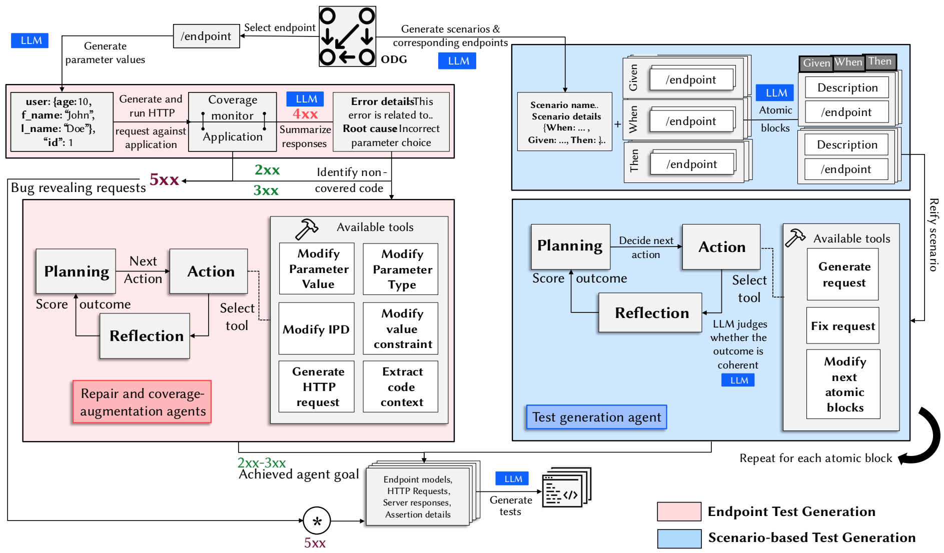

Figure 6: The workflow for generating endpoint-focused and scenario-based tests.

All the tools (or actions), except the code-context action, generate one or more HTTP requests. The code-context action produces code-related details returned from the CLDK API chosen. During reflection for this action, the agent evaluates whether it contributes meaningfully to facilitating future request generation in subsequent iterations. As part of reflection, the agent computes a score in the range [0, 1], indicating the effectiveness of the action, and creates a comment explaining the rationale for the score. These are used in the next iteration to guide the selection of the next action.

3.3.5 Test generation step. In this step, we convert selected requests into executable test cases. Specifically, we focus on two types of requests: (a) those that contribute to code coverage and (b) those that reveal potential bugs, such as requests resulting in 5xx response codes. For identifying coverage-contributing requests, we rely on a coverage monitoring agent deployed alongside the application. Once these requests are identified, we extract endpoint information and generate corresponding test cases using a code skeleton defined through Jinja templates [49]. We ensure that the generated tests include appropriate package declarations and are organized according to the application's source code directory structure. This design choice aids developers in easily mapping each test to its corresponding endpoint class, improving traceability and maintainability.

3.4 Scenario-based Test Generation

Scenario-based test generation focuses on creating meaningful sequences of API calls for exercising application use cases, as illustrated by the PetClinic test case in Listing ??. The right side of Figure 6 illustrates the workflow for generating scenario-based tests, which consists of four steps. In the first step, SAINT extract test scenarios from the application code and map each scenario to a sequence of endpoints via an LLM call. The second step decomposes a scenario into a sequence of atomic blocks, where each atomic block achieves a specific step of a test scenario and is associated with one endpoint. In the third step, SAINT employs an agentic approach to generate the test fragment for each atomic block, using

```
scenario: View list of veterinaries and visit a selected veterinary
given:
  - A vet with ID 1 exists in the system
  - An owner with ID 1 exists in the system having a pet with ID 1
when:
  - A request is made to the 'showVetList' endpoint with vet ID 1
  - The vet's details are viewed
  - A request is made to 'initNewVisitForm' to initialize a new
    visit form
  - The form is filled and submitted to 'processNewVisitForm'
then:
  - The vet's information is displayed
  - The new visit is added to the pet's record in the database
```

Figure 7: Sample test scenario extracted by SAINT.

the generated information for a block to process subsequent blocks. The final step composes the test fragments together to create an executable JUnit test case for the scenario.

3.4.1 Generation of test scenarios and related endpoints. To generate test scenarios, our approach constructs an LLM prompt that includes endpoint information and functional summaries. The prompt instructs the model to produce scenarios aligned with business use cases using Gherkin-like syntax [12]. Each scenario follows a structured format with a scenario name, a `given` clause (preconditions), a `when` clause (actions), and a `then` clause (expected outcomes). We experimented with various formats and found the Gherkin-like style most effective, even enabling smaller models to generate coherent, meaningful scenarios. Figure 7 shows an example generated for the PetClinic application [44].

3.4.2 Decomposing scenarios into atomic blocks. Each clause of a scenario can be associated with one or more endpoints. If a `given`, `when`, or `then` clause has more than one endpoint, SAINT decomposes it into more granular tasks, or *atomic block*, so that processing a single endpoint can help achieve that task. The prompt instructs the LLM to divide a scenario into atomic blocks and provide details on how the blocks are related. For the example scenario in Figure 7, the `when` can be divided into more granular tasks of retrieving vet information, initializing a new visit form, and creating a new visit, where the last two tasks share the same pet and owner ID. The information about decomposed blocks is then fed to the test-generation agent to create concrete HTTP requests and test case.

3.4.3 *Test-generation agent.*

The agent processes each atomic block of a scenario to generate concrete requests, and uses the responses from those requests to process the subsequent blocks. In the planning step, the agents decide on the next course of action and selects from a set of available tools. We designed three tools tailored to the task of converting a sequence of atomic blocks for a scenario to a sequence of test fragments.

(1) *Generate requests.* With this tool, the agent generates HTTP requests for the endpoint corresponding to an atomic block. The relevant context in the LLM prompt includes endpoint details, the scenario description, the task for the particular block, the outcomes of preceding blocks (requests and corresponding responses), and in-content examples. The generated requests are executed against the deployed application to obtain the responses.

(2) *Fix request.* With this tool, the agent attempts to modify the parameter values of a previously generated request so that the request aligns with the task description for an atomic block.

(3) *Modify subsequent atomic blocks.* The outcome of one block can require modifications to subsequent blocks. Consider again the scenario in Figure 7. Suppose that while processing the task for the `given` clause, the agent finds that there exists a vet with ID 2; this ID can also be used for the scenario, but it requires the descriptions of the subsequent blocks to be updated. The agent uses the scenario description, the current block being processed, and its requests and outcomes to modify descriptions of the subsequent blocks (if required) such that the overall goal of the scenario is preserved while minor details (e.g., vet ID) are updated.

After executing an action, the agent reflects on the outcome of the executed requests to determine whether it aligns with the description of the current block under processing. If it determines that the requests are unrelated to the block, it provides a justification for its decision. If the outcome aligns, the agent also determines whether any subsequent blocks in the scenario need modifications. The information from this step then feeds into the next iteration of planning to select action to be performed.

3.4.4 *Generate Tests.*

After the test-generation agent completes, SAINT collects all relevant information—including the scenario description, the atomic block sequences, the HTTP requests, and the responses—and prompts an LLM to generate a test using the Rest-Assured framework [41]. Given the structured nature of this data and the simplicity of Rest-Assured's syntax, the LLM consistently produces compilable tests. Finally, SAINT appends the scenario description and Java package information to produce the complete scenario-based test.

4 Evaluation

Our evaluation focuses on the following five research questions:

- **RQ1**: (Coverage) How does SAINT compare with EvoMaster [2] in terms of code coverage, operation coverage, and database interaction coverage achieved?
- **RQ2:** (Scenario Effectiveness) How effective is SAINT in generating scenario-based tests?

Table 1: Java applications used in the evaluation.

Dataset	Framework	Java version	OpenAPI spec?	NCLOC	# of classes	# of endpoints
DayTrader	Servlet	8	X	11409	141	113
PetClinic	Spring	17	X	790	24	17
JPetStore	Stripes	8	X	1409	24	21
Restcountries	Jax-rs	8	✓	1619	23	27
Feature-service	Jax-rs	8	✓	1688	21	18
Genome-Nexus	Spring	8	✓	22143	74	48
Languagetool	HttpServer	8	✓	113170	37	6
App X	Servlet	11	X	1255	24	23

- **RQ3:** (Developer survey) How do developers perceive the scenario-based tests generated by SAINT in terms of their usefulness?
- **RQ4:** (Fault Triggering) How does SAINT compare with EvoMaster in terms of server failures triggered?
- **RQ5:** (Ablation) How do ODG construction, IPDs and value constraints extraction, repair agent, and coverage-augmentation agent contribute to SAINT's effectiveness in code coverage?

4.1 Experiment Setup

We evaluated our approach on two categories of service-oriented applications: (1) REST APIs with OpenAPI specifications and (2) enterprise Java applications without OpenAPI specifications. The first group includes four applications (Feature-Service, Genome Nexus, LanguageTool, and RestCountries) from the EvoMaster benchmark [55]. The second group comprises three open-source Java applications (DayTrader [10], JPetStore [18], and PetClinic [39]) and one proprietary enterprise application. These three open-source applications were also analyzed in prior studies [35, 38], which examined six Java EE applications; we added three applications in our evaluation dataset after excluding those that could not be deployed or crashed frequently. Table 1 shows the dataset characteristics.

Testing Tools. We compare SAINT with EvoMaster, a state-of-the-art white-box test generation tool [2]. Although EvoMaster operates in white-box mode, it still requires an OpenAPI specification as input. As a result, our comparison is limited to applications for which OpenAPI specifications are available. For the remaining applications in our dataset, we evaluated several off-the-shelf specification generation tools (e.g., springdoc-openapi [45], SpringFox [40]); however, these tools consistently produced incomplete specifications and required substantial manual effort to make them usable with EvoMaster. In the case of EvoMaster, each application was executed for one hour using a randomly chosen seed. In white-box mode, EvoMaster attaches its own JVM agent to instrument the application under test, which is known to conflict with the JaCoCo agent [56]. Although EvoMaster also reports code coverage, its measurement approach differs from that of JaCoCo. To ensure a reliable comparison, we first used EvoMaster to generate tests for each application. We then executed those generated tests with the JaCoCo agent enabled, allowing us to collect consistent and comparable code-coverage data across all the applications.

LLMs. We selected models based on size, cost, model family, and popularity, categorizing them as small (IBM Granite 3.1–8B, Meta Llama 3.1–8B), medium (Devstral-24B, DeepSeek-R1-distill-Qwen-32B), and large (GPT o1). Due to the high computational cost of evaluating multiple models across datasets, each was run twice using parameters from prior work [38], with a temperature setting of 0.2 to ensure stable yet diverse outputs. SAINT uses 25 unique prompts in its pipeline (these are available in our artifact [49]).

Metrics. To evaluate our approach, we used various coverage and quality metrics. For code coverage, we measured line and branch coverage, as well as database line coverage, using CLDK [8] to identify database interaction points. To measure code coverage, we use the JaCoCo agent [14], attaching it to the running application instance to record coverage in real time as requests are executed. We also computed reachability coverage by analyzing static call chain starting from each endpoint and identifying all reachable methods, which let us measure coverage within the effective execution scope—the portion of the code that is statically reachable from the endpoint. In addition, we report operation coverage, representing the proportion of distinct API operations (i.e., the combination of endpoint resource path and HTTP operation or an endpoint method) exercised. For scenario quality, we computed scenario length, number of scenarios, and other structural indicators.

4.2 Experiment Results

4.2.1 RQ1: Code and operation coverage of individual endpoint test generation. In this RQ, we evaluate SAINT's capability to generate endpoint-focused tests against two sets of applications: those with and without OpenAPI specifications. Figure 8 presents the results.

Applications without OpenAPI Specifications. In this category, we analyzed four applications—three open-source projects (PetClinic, DayTrader, and JPetStore) and one closed-source application (App-X). SAINT was able to capture database coverage for all except App-X, which uses DB2—a database currently unsupported by CLDK. In terms of database line coverage, the o1 model performed best overall. However, for PetClinic and JPetStore, all models achieved comparable coverage. A manual inspection of JPetStore identified four database call sites, three of which involve complex conditional logic—posing challenges for all models. Interestingly, in Spring-PetClinic, all models achieved similar line, branch, and database coverage. Further analysis revealed that gaining additional coverage would require solving intricate constraints, which SAINT currently does not support. Overall, model performance was comparable, with the smaller Granite-8B often matching or outperforming larger models like o1. The generated tests are implemented in Java using the Rest-assured framework [41]. An example is shown below:

```
@Test
public void testProcessUpdateForm1() {
    Map<String, String> pet = new HashMap<>();
    pet.put("name", "Fido");
    pet.put("birthDate", "2010-05-01");
    given()
        .queryParam("pet", pet)
        .pathParam("ownerId", "1")
        .pathParam("petId", "1").when()
        ↪   .post("http://localhost:8080/owners/{ownerId}/pets/{petId}/edit")
    .then()
        .statusCode(302);}
@Test
public void testProcessUpdateForm2() {
    Map<String, String> pet = new HashMap<>();
    pet.put("name", "Buddy");
    pet.put("birthDate", "2012-08-15");
    given()
        .queryParam("pet", pet)
        .pathParam("ownerId", "10")
        .pathParam("petId", "2").when()
    .post("http://localhost:8080/owners/{ownerId}/pets/{petId}/edit")
    .then()
        .statusCode(500); } ...}
```

The tests simulate a user updating the details of an existing pet. SAINT generates multiple tests covering both positive and negative paths. In the first scenario, it uses valid owner and pet IDs to update pet information, resulting in a successful 200 response. In contrast, the second scenario uses invalid (non-existent) IDs, which leads to a 500 server error. All tests generated by SAINT are directly compilable and executable without requiring manual edits.

> **Finding 1:** SAINT can generate tests for applications without OpenAPI specifications with high code coverage (10%–80%). Also, with SAINT, smaller models (8B parameters) can achieve similar or better coverage compared to bigger models such as GPT-o1.

Applications with OpenAPI Specification. Among all the applications under this category, Restcountries and Languagetool do not interact with any database, while Genome-Nexus uses MongoDB, which is currently unsupported by CLDK. Consequently, database coverage is only reported for Feature-service. In this case, SAINT achieves notably higher database coverage, successfully exercising multiple database interaction points. We also found that SAINT can identify more endpoints than those defined in the OpenAPI specification—most notably in Genome-Nexus, where the specification lists 23 endpoints, but SAINT detected 48. In terms of line and branch reachability coverage, SAINT outperforms EvoMaster for Feature-service and Genome-Nexus (+50.5% and + 22.0% in line coverage). For Restcountries, SAINT's performance is slightly lower (-0.9% in branch coverage). However, for Languagetool, SAINT's performance is worse compared (-19.3%) to EvoMaster in case of application coverage. However, when compared on reachability coverage, they are comparable. The discrepancies observed in LanguageTool arise from an incomplete API specification: the OpenAPI specification lists only two endpoints, while more endoints exist in the implementation. This is not uncommon, as OpenAPI specifications are primarily written for external users and often omit internal endpoints.

> **Finding 2:** Compared to EvoMaster, SAINT achieves similar or considerably better coverage, with the coverage difference ranging from -0.9% to +50.5%.

4.2.2 RQ2: Effectiveness of scenario-based test generation. Besides testing individual endpoints, a key feature of SAINT is extracting test scenarios and converting them into Java tests. First, SAINT employs an LLM to generate a brief summary of each endpoint's behavior. We use the ODG to extract and convert application use cases into Java tests. To assess their quality, we evaluate the number, length, endpoint class diversity, and good-path versus bad-path distribution of scenarios, as shown in Table 2. We classify scenarios as good-path if they result in a 2xx status code, and as bad-path if they result in a 5xx status code. On average, the LLM generated 6, 11, 7, 8, 10, 10, and 9 scenarios for PetClinic, DayTrader, JPetStore, Feature-service, RestCountries, Genome-Nexus, and App-X, respectively. No scenarios were generated for LanguageTool, which contains only one valid endpoint. Most LLM-generated scenarios span multiple endpoints, with larger models like o1 generally producing longer sequences—observed in 6 of the 7 applications. Notably, 42.6% of scenarios span multiple endpoint classes, underscoring the need for

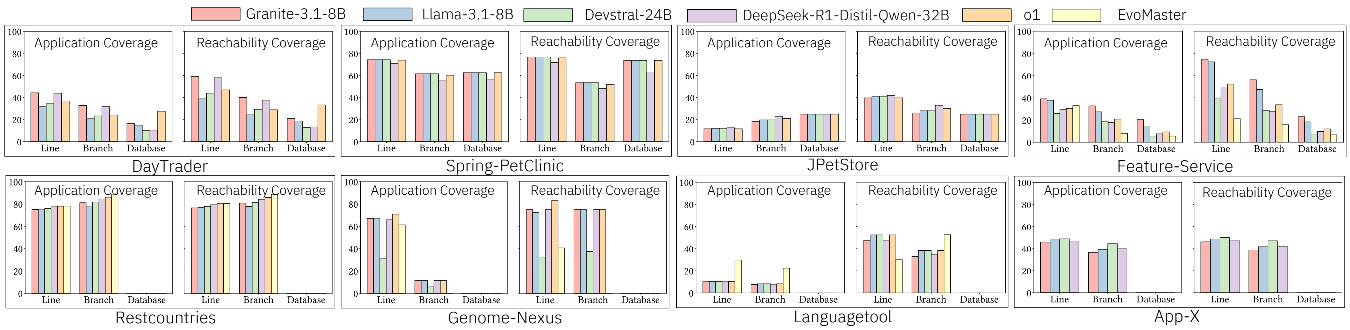

Figure 8: Application and reachability coverage.

Table 2: Effectiveness of scenarios and scenario-based tests.

App	Models	# of scenarios	Sequence length	Scenarios w/ >1 class (%)	Coverage (%) Line	Branch	DB	Operation	(good, bad) path scenario (%)
PC	G-8B	4.0	2.8	87.5	57.5	41.7	47.5	52.9	(87.5, 12.5)
	L-8B	7.0	2.4	14.3	64.3	53.2	47.9	61.8	(71.4, 28.6)
	DV-24B	7.0	2.3	5.6	68.4	48.1	50.5	82.4	(78.9, 21.1)
	DS-32B	4.5	2.4	32.5	61.6	50.6	55.5	52.9	(90.0, 10.0)
	o1	6.0	3.2	50.0	61.2	37.2	42.1	85.3	(66.7, 33.3)
DT	G-8B	6.5	1.5	18.1	7.8	6.4	7.4	23.2	(100.0, 0.0)
	L-8B	8	2.4	60.0	8.1	6.9	5.6	25.6	(90.0, 10.0)
	DV-24B	18	2.1	25.3	8.0	6.4	7.4	43.9	(68.2, 31.8)
	DS-32B	14	2.0	13.9	8.1	6.4	7.4	42.7	(92.2, 7.8)
	o1	8	4.2	62.5	10.2	10.1	5.6	80.5	(93.7, 6.3)
JP	G-8B	4.5	4	90.0	7.8	5.3	-	45.2	(35.0, 65.0)
	L-8B	10	2.5	31.2	8.0	5.3	-	69.0	(43.8, 56.2)
	DV-24B	10	4.4	65.0	8.1	5.3	-	97.2	(25.0, 75.0)
	DS-32B	7.5	3.2	42.0	8.1	5.3	-	85.7	(45.5, 54.5)
	o1	5	5.2	20.0	8.1	5.3	-	100	(60.0, 40.0)
FS	G-8B	5.5	2.6	60.7	21.2	11.3	0.0	61.1	(21.4, 78.6)
	L-8B	14.5	2.3	46.1	26.6	16.1	3.7	80.6	(28.4, 71.6)
	DV-24B	11.5	2.0	31.5	19.0	9.5	1.5	83.3	(20.4, 79.6)
	DS-32B	4.0	5.9	70.0	13.4	8.3	5.9	77.8	(36.7, 63.3)
	o1	3.5	4.9	70.0	36.5	33.9	19.8	75.0	(65.0, 35.0)
RC	G-8B	5.0	2.1	10.0	38.7	30.0	-	33.9	(70.0, 30.0)
	L-8B	11.0	3.0	20.0	30.0	20.2	-	26.8	(29.2, 70.8)
	DV-24B	15.5	1.5	12.5	63.8	63.6	-	51.8	(61.2, 38.8)
	DS-32B	10.5	1.4	16.7	44.9	43.6	-	42.8	(59.7, 40.3)
	o1	7.5	3.9	100.0	65.4	59.3	-	94.6	(80.0, 20.0)
GN	G-8B	7.5	1.9	81.3	49.0	1.9	-	19.8	(72.3, 27.7)
	L-8B	13.0	1.7	0.0	54.2	1.9	-	45.83	(69.3, 30.8)
	DV-24B	14.0	1.8	36.4	53.1	7.7	-	41.7	(74.6, 25.4)
	DS-32B	6.0	2.3	95.0	49.1	1.9	-	19.8	(100.0, 0.0)
	o1	8.0	2.5	25.0	54.5	7.7	-	39.6	(75.0, 25.0)
A-X	G-8B	8.0	2.2	48.4	24.1	11.6	-	28.3	(56.3, 43.7)
	L-8B	16.5	2.7	73.3	27.9	14.4	-	56.5	(61.7, 38.3)
	DV-24B	10.0	2.2	35.0	30.9	18.1	-	50.0	(65.0, 35.0)
	DS-32B	4.0	3.1	80.0	26.0	13.9	-	32.6	(90.0, 10.0)

* G-8B: Granite-3.1.-8B, L-8B: Llama-3.1.-8B, DS-32B: DeepSeek-R1-Distil-Qwen-32B, DV-24B: Devstral Small, PC: PetClinic, DT: DayTrader, JP: JPetStore, FS: Feature-Service, RC: Restcountries, GN: Genome-Nexus, A-X: App-X

```java
public class ProductCreationWithFeaturesConstraintsAndConfigurationTest {
    @Test @Order(1)
    void createNewProductLaptop() {...
        .post("/products/Laptop")...}
    @Test @Order(2)
    void addFeatureTouchScreenToLaptop() {...
        .post("/products/Laptop/features/TouchScreen")...}
    @Test @Order(3)
    void addFeatureStylusToLaptop() {...
        .post("/products/Laptop/features/Stylus") ...}
    @Test @Order(4)
    void addFeatureFingerprintScanToLaptop() {...
            .post("/products/Laptop/features/FingerprintScan")...}
    @Test @Order(5)
    void addRequiresConstraintBetweenTouchScreenAndStylus() {...
            .post("/products/Laptop/constraints/requires")...}
    @Test @Order(6)
    void addExcludesConstraintBetweenStylusAndFingerprintScan() {...
            .post("/products/Laptop/constraints/excludes") ...}...}}
```

more than isolated endpoint testing. As an example, a scenario generated by o1 for Feature-service begins with adding a new product (`Laptop`), followed by features (`TouchScreen`, `Stylus`), setting constraints, and adding multiple configurations. The complete code for this scenario is provided in the supplementary material [49].

```java
String variant = "7:g.140453136A>T";
Response response = ...
        .get("/cancer_hotspots/hgvs/{variant}", variant)
        ....
String genomicLocation = "7:140453136";
Response response = ...
        .get("/cancer_hotspots/genomic/{genomicLocation}", genomicLocation)
String[] variants = {"7:g.140453136A>T", "12:g.25398285C>A"};
Response response = ...
        .post("/cancer_hotspots/hgvs")
```

Finding 3: Larger models tend to focus on more complex scenarios in an application, generating scenarios with longer sequence.

Scenario code coverage. In terms of code coverage, scenario-based test generation does not achieve the same level of performance as individual endpoint testing. While scenario-based test generation is well-suited for validating complex, end-to-end use cases, individual endpoint testing remains more effective for maximizing code and operation coverages. For the DayTrader application, where both code and operation coverage were particularly low during scenario-based test generation. This is largely due to the nature of the application's endpoints—many of which are designed to send pings to various services or perform status checks, making them unsuitable candidates for meaningful scenario construction.

Finding 4: While scenario-based test generation is well-suited for validating complex application use cases, individual endpoint testing is more effective for maximizing code coverage.

Another interesting observation is that these scenarios can go beyond the conventional endpoint dependencies. For example, in Genome-Nexus, we observed a scenario related to fetching and analyzing cancer hotspot annotations. This scenario involves three endpoints that are not directly connected through traditional endpoint relationships. However, the endpoints are semantically related through the variant and genome location they reference. This indicates that an LLM-based approach can detect functional relationships beyond explicit structural dependencies.

4.2.3 RQ3: Developer's preference about scenario-based test generation.

We conducted a survey on 41 employees of organization X (removed for anonymity) to get the qualitative feedback on the test scenarios and corresponding tests generated by SAINT.

Survey design. Table 3 presents the survey questionnaire. Broadly, it is divided into four major sections.

Professional Background. This section presents seven questions about participants' professional backgrounds, covering their roles,

Table 3: Survey questionnaire.

Type	Question	Format
Professional Background	Q1. Current Role	Open
	Q2. Years of experience in software engineering (incl. education)	MCQ
	Q3. Years of experience in industry	MCQ
	Q4. Level of expertise in Java	MCQ
	Q5. Prior experience with automated API-level test generator	MCQ
	Q6. How often do you write API-level tests for your codebase?	MCQ
	Q7. On average, how long do you spend writing API-level tests for an application use case (i.e., tests that exercise a functional scenario)?	MCQ
Scenario Quality	Q8. I understand what this scenario describes	Likert
	Q9. The scenario covers a meaningful functionality of the application	Likert
	Q10. Testing such scenarios is valuable for validating the application	Likert
	Q11. I would test such scenarios if this were my application under test	Likert
Generated Test Quality	Q12. I understand what the test does	Likert
	Q13. The test is well structured	Likert
	Q14. The test feels natural (in terms of variable, method, and class names)	Likert
	Q15. This test correctly implements the test scenario	Likert
	Q16. The input values (e.g., string literals, integer constants), if any, used in the test case are meaningful	Likert
	Q17. The test sequence (i.e., the sequence of API calls) makes sense	Likert
	Q18. The test assertions are meaningful	Likert
	Q19. Would you add the test case to your service-level test suite?	MCQ
Comment	Q20. What are your thoughts on the strengths and weaknesses of the extracted scenarios and the generated tests? Do you believe they are suitable as outputs from a fully automated process?	Open

experience in SE and Java, and involvement in API testing. It also assesses their familiarity with automated API test generators, the frequency of API testing tasks, and the time typically spent writing high-quality tests involving multiple API calls.

Test scenario quality. For this section, we selected three relatively easy-to-understand applications—PetClinic, Feature-service, and App-X. Participants could select one of the applications based on their preference. They were then shown three LLM-generated test scenarios with corresponding Java code. The scenarios were selected from top-performing model–application pairs (as identified in RQ2): o1 for PetClinic and Feature-service, and DeepSeek-R1-32B for App-X. From multiple runs, three scenarios were randomly sampled for each case. Contextual information about the application and relevant endpoints was provided to aid comprehension. Participants evaluated each scenario based on understandability, functional relevance, and whether they would include it in the test suite for the application.

Generated test quality. For each test scenario, participants reviewed the corresponding generated code and assessed its clarity and quality. They evaluated whether the test's purpose was clear, the structure logical, and naming conventions (e.g., variables, methods, classes) felt natural. They also judged whether the test accurately implemented the described scenario, whether the API call sequence was meaningful, and whether the assertions were appropriate and relevant.

Overall feedback. In this section, participants could provide more detailed feedback regarding the strengths and weaknesses of SAINT, as well as suggestions for tool improvement.

Participant background. Participants in the survey came from diverse professional roles, including developers, architects, QA engineers, product managers, and researchers. Most had a strong software engineering background—79% reported over 15 years of experience (including education), and 66% had more than 10 years in industry. Additionally, 76% had Java expertise, and 78% actively performed API testing. Notably, 42% said writing high-quality API tests takes over 30 minutes. More than half of the participants responded that they never used any automated test generation tools. These findings underscore both the participants' qualifications and

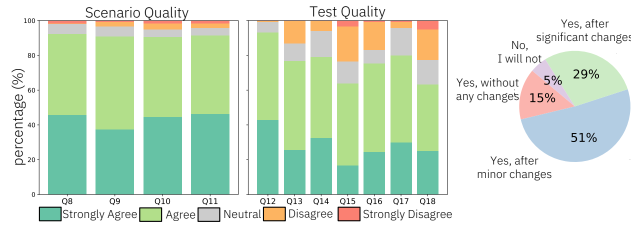

Figure 9: Scenario and test quality assessment (left) and test acceptance (right) by developers.

the potential productivity gains from generating high-quality API tests. Full details are available in the supplementary material [49].

> **Finding 5:** 42% of the participants indicated that writing high-quality API tests is time-consuming, often requiring more than 30 minutes to complete.

Test scenario quality. For test scenarios, participants reviewed the natural language descriptions and answered questions about their quality and relevance. The feedback indicated that SAINT-generated scenarios were highly rated for understandability, meaningfulness, and alignment with real application use cases. Over 90% of participants agreed that they would test similar scenarios, while fewer than 5% expressed a preference against using them.

> **Finding 6:** More than 90% participants agreed that they would test application scenarios similar to the ones extracted automatically by SAINT.

Generated test quality. In this phase, developers evaluated the reified test scenarios and the corresponding generated code, focusing on understandability, formatting, and naturalness—particularly in method, class, and variable names, as well as input values. They also assessed the clarity of test sequences, the quality of assertions, and alignment with the described scenarios. Over 70% of participants found the tests easy to understand and natural. However, more than 20% noted room for improvement in assertion quality and scenario alignment. Still, over 60% agreed the tests were correctly implemented and included meaningful assertions. Notably, 66% indicated they would include the generated tests in their suite with minor or no changes, 29% would do so with significant modifications, and only 5% would not use them at all (reported in Figure 9).

> **Finding 7:** Participants responded positively to various aspects of the generated tests, with approximately 66% indicating that they would add the tests to their regression test suites with little to no modification.

Overall strengths and weaknesses. Participant feedback highlighted one of SAINT's key strengths: its ability to generate well-structured and readable tests. As one participant noted, "there are things to make it easier to read… like a central definition of certain variables (BASE_URI)." Several participants also suggested improvements to enhance the practicality of SAINT. A common request was to make tests more self-contained. Currently, tests rely on hardcoded values and modify application state without handling setup or cleanup. High-quality tests, however, should create necessary resources at runtime and perform cleanup afterward—an important direction for future development. Another suggested improvement was enhancing assertion quality. Currently, SAINT generates assertions

Table 4: Fault detection.

Application	Granite-8B	Llama-8B	Devstral	DeepSeek	o1	EvoMaster
DayTrader	10	2	7	6	2	-
Spring-PetClinic	4	5	5	4	10	-
JPetStore	9	7	6	6	6	-
Feature-service	76	179	67	65	81	681
Restcountries	1	1	1	1	1	1
Genome-Nexus	0	0	0	0	0	0
Languagetool	0	0	0	0	0	9
App-X	0	0	0	0	-	-

based on response codes and raw server output. Participants noted that validating more success and failure paths—would significantly improve the tests' effectiveness.

4.2.4 RQ4: SAINT's effectiveness in triggering faults. We evaluated SAINT 's ability to trigger faults using the methodology from prior work [22]. We also reused the regex patterns from a prior work [33] to identify unique request–response pairs. To improve generality, we enhanced the implementation so it can parse requests and responses and automatically learn new regex patterns for unseen cases. We found that in most cases, SAINT mostly found faults in applications, except for Genome-Nexus and App-X, which likely did not show faults due to lack of 5xx errors. SAINT triggered more faults compared to EvoMaster for Feature-service and RestCountries applications but not able to trigger any failure for LanguageTool. The high number of faults in the Feature-service occurs because it returns an entire HTTP page as the response—along with the stack trace—whenever a fault is triggered. This makes the errors difficult to distinguish using simple string-based operations.

4.2.5 RQ5: Effectiveness of SAINT components. Our ablation study evaluates the key components of SAINT. Due to the high computational demands of testing multiple models, we chose Granite-8B, our smallest effective model and performed the study on four applications with and without OpenAPI: DayTrader, PetClinic, Feature-service, and Genome-Nexus. Table 5 presents the ablation results. Partial orderings from the ODG had the greatest positive effect on test coverage, followed by IPD extraction, value constraint extraction, the coverage-augmentation agent, and the repair agent.

5 Discussion

Cost of using SAINT. One major concern with LLM-based approaches is their operational cost, as excessive token consumption can substantially increase overall expenses and limit practical usage. To evaluate this aspect, we measured the token usage for both endpoint-level and scenario-based test generation, and translated these values into monetary cost using the pricing of different models. Our analysis demonstrates that SAINT is significantly cost-efficient, both in terms of total expenditure and the number of LLM invocations. Specifically, with Devstral (via OpenRouter [37]) the average cost is \$0.24 for endpoint-focused and \$0.22 for scenario-based generation, whereas for o1 (on the OpenAI platform), it is \$6.17 and \$4.42, respectively. The detailed result in present in the supplementary material [49]. We believe this efficiency primarily because of the integration of static analysis, which enables SAINT to perform targeted pre-processing and invoke the LLM only when necessary, rather than feeding the entire application context to an off-the-shelf model.

Benefits and drawbacks of using hybrid approach. Beyond cost, hybrid approaches offer several additional advantages. For example, in terms of HTTP call efficiency, SAINT performs significantly fewer requests compared to EvoMaster. On average, SAINT issues fewer than 500 HTTP requests per application, whereas EvoMaster generates over 10k+ requests for the same applications, resulting in a more focused and efficient testing process. Another key advantage is scalability—SAINT can support major Java versions and frameworks, while conventional tools often struggle to maintain compatibility as the scope expands. Although SAINT continues to rely on static analysis, it leverages well-established and actively maintained analysis tools that provide broad language support.

6 Related Work

Automated REST API testing techniques are categorized into black-box and white-box approaches [29, 34]; former relies on API specs and the latter on source code inspection and runtime monitoring.

Black-box API testing. Early black-box techniques use fuzzing and model-based strategies to generate request sequences from the OpenAPI Specification. Tools such as RESTler [4], RestTestGen [52], MoREST [28], and RAFT [42] conduct stateful fuzzing via graph traversals and HTTP method differentiation. Recent LLM-enhanced tools, like KAT [26] and LogiAgent [54](arXiv), infer semantic relationships, outperforming heuristic methods. Other LLM-augmented API testing systems, including RESTGPT [23], AutoRestTest [24, 46], and LlamaRestTest [22], improve parameter generation and IPD extraction. AutoRestTest and LogiAgent also use agents for iterative test refinement. However, these methods are limited by inaccuracies in OpenAPI specifications [11, 30], affecting their use in poorly documented real-world applications.

White-box API testing. White-box techniques use internal system knowledge to enhance coverage and fault detection. EvoMaster [2] applies evolutionary algorithms based on code coverage and mutations, while MioHint [27] uses LLMs to improve inputs for challenging branches through static analysis. However, both face difficulties in creating semantically meaningful request sequences.

Scenario-based testing. Scenario-based testing simulates real-world workflows through dependent API sequences. Traditional black-box tools, such as RESTler [4], MoREST [28], and RestTestGen [52], infer dependencies heuristically. Recent tools like RAFT [42], KAT [26], and LogiAgent [54] integrate LLMs, with LogiAgent generating scenarios from endpoint descriptions. However, these approaches rely solely on OpenAPI specs and overlook hidden code-level and database dependencies.

Positioning SAINT. SAINT is the first white-box, LLM-based agentic framework for REST API testing. It extends model-based techniques (e.g., RESTler, RestTestGen, KAT) by leveraging static analysis to identify dependencies and sequence operations beyond the OpenAPI specification, enabling accurate, meaningful scenarios. End-to-end LLM integration generates semantic tests that surpass fuzzing-based white-box methods like EvoMaster. Compared to LogiAgent—the closest counterpart—SAINT incorporates code understanding for request sequencing and generation, overcoming reliance on incomplete external resources. Moreover, SAINT's comprehensive tool-chain for static analysis, IPD extraction, and generation enhances autonomy and adaptive reasoning.

Table 5: Results of the ablation study.

Coverage Metric		Value constraints	IPD	Partial order	Request fixing	Coverage augmentation
Application	Line	+9.0	+12.0	+17.1	+2.2	+0.4
	Branch	+33.2	+26.0	+34.2	+8.4	+13.6
	Database	+7.2	+35.0	+14.0	-1.3	+5.2
Reachability	Line	+15.4	+17.0	+27.0	+5.5	+1.9
	Branch	+31.7	+31.0	+36.6	+12.8	+14.9
	Database	+5.7	+38.0	+16.3	-1.2	+6.3

REST API specification generation. To address missing or incomplete application specifications, several techniques have been proposed to automatically generate REST API specifications for use in test generation. Commercial tools such as SpringFox [40], springdoc-openapi [45], and Swagger Core [50] generate OpenAPI specifications through runtime inspection and reflection, but they are tied to specific frameworks and often produce incomplete specifications that require substantial engineering effort to integrate with test generators. Research tools like Respector [13] provide higher-quality specifications but support only Spring and JAX-RS applications on specific Java versions. In contrast, SAINT requires no OpenAPI specification and goes beyond testing individual endpoints.

7 Threats to Validity

To evaluate SAINT, we used line and branch coverage with server-side error detection. However, these metrics may not fully capture the application's behavioral and business validity. Therefore, our test generation includes scenario-based tests that replicate functional workflows. We used both quantitative and qualitative measures to assess the quality of these scenarios, including developer feedback via a survey.

The use of LLMs and agents derived from LLMs exposes SAINT to an inherent stochasticity and sensitivity to prompt structure that may affect repeatability of our results. To limit this, we used temperature as low as 0.2 and performed two runs with each LLM. However, to mitigate the impact of running fewer trials on the reported coverage, we repeated the experiment with Devstral (due to its lower inference cost) ten times. We found that the standard deviation of both line and branch coverage was very low (for branch coverage, 0.0–5.2% for individual endpoints and 0.0–7.6% for scenario-based tests; for line coverage, 0.1–1.3% for individual endpoints and 1.4–5.1% for scenario-based tests). These results are available in our artifact [49].

Our evaluation included eight applications built on different frameworks with and without OpenAPI specifications. We specifically target REST API synchronous request/response models within Java. The results may not extend to applications that employ different communication methods (e.g., message queues or event-driven architectures) or to applications implemented in other languages. Another limitation of SAINT, similar to EvoMaster, is its inability to mock external services. We plan to address these shortcomings in future work.

8 Conclusion

We presented SAINT, a white-box approach for service-level testing of enterprise Java applications combining static analysis with LLM-driven agentic workflows. SAINT generates both endpoint-focused and scenario-based test cases, targeting high code coverage and realistic use-case execution. Our technique integrates symbolic information extracted through static analysis with semantic reasoning capabilities of LLMs, supported by agents that repair, augment, and compose tests. Evaluation across eight applications demonstrates that SAINT outperforms prior approaches in test coverage and scenario realism, with favorable feedback from developers and supporting ablation analyses. We identify several future research directions: extending SAINT to support more backends (e.g., Python Flask, Node.js Express) and interfaces (e.g., GraphQL, gRPC), improving the generated scenario-based tests to be self-contained, and developing human-in-the-loop variants of SAINT for interactive scenario generation and domain-specific test refinement.

References

[1] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*. 794–807.

[2] Andrea Arcuri. 2018. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 394–397. doi:10.1109/ICST.2018.00046 arXiv:1901.04472 [cs].

[3] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1, Article 3 (jan 2019), 37 pages. doi:10.1145/3293455

[4] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 748–758. doi:10.1109/ICSE.2019.00083

[5] David F Bacon and Peter F Sweeney. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 324–341.

[6] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. 2009. WS-TAXI: A WSDL-based Testing Tool for Web Services. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*. IEEE Computer Society, 326–335. doi:10.1109/ICST.2009.28

[7] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2024. Random Testing and Evolutionary Testing for Fuzzing GraphQL APIs. *ACM Trans. Web* 18, 1, Article 14 (Jan. 2024), 41 pages. doi:10.1145/3609427

[8] cldk 2025. CodeLLM-Devkit. https://github.com/codellm-devkit/python-sdk

[9] Davide Corradini, Zeno Montolli, Michele Pasqua, and Mariano Ceccato. 2024. DeepREST: Automated Test Case Generation for REST APIs Exploiting Deep Reinforcement Learning. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 1383–1394. doi:10.1145/3691620.3695511

[10] daytrader 2025. DayTrader8 Sample. https://github.com/OpenLiberty/sample.daytrader8

[11] Sida Deng, Rubing Huang, Man Zhang, Chenhui Cui, Dave Towey, and Rongcun Wang. 2025. LRASGen: LLM-based RESTful API Specification Generation. *arXiv preprint arXiv:2504.16833* (2025).

[12] gherkinsyntax 2025. Gherkin Reference. https://cucumber.io/docs/gherkin/reference

[13] Ruikai Huang, Manish Motwani, Idel Martinez, and Alessandro Orso. 2024. Generating REST API Specifications through Static Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Article 107, 13 pages. doi:10.1145/3597503.3639137

[14] JaCoCo. 2025. JaCoCo Agent. https://www.eclemma.org/jacoco/trunk/doc/agent.html. [Online; accessed Nov-2025].

[15] jakarta 2025. Jakarta EE. https://jakarta.ee/

[16] JavaParser. 2025. JavaParser. https://github.com/javaparser/. [Online; accessed Nov-2025].

[17] jdkhttpserver 2025. JDK HttpServer. https://docs.oracle.com/en/java/javase/21/docs/api/jdk.httpserver/com/sun/net/httpserver/HttpServer.html

[18] jpetstore 2025. MyBatis JPetStore. https://github.com/mybatis/jpetstore-6

[19] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2021. Automatic Property-based Testing of GraphQL APIs. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. 1–10. doi:10.1109/AST52587.2021.00009

[20] Myeongsoo Kim, Davide Corradini, Saurabh Sinha, Alessandro Orso, Michele Pasqua, Rachel Tzoref-Brill, and Mariano Ceccato. 2023. Enhancing REST API Testing with NLP Techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, 1232–1243. doi:10.1145/3597926.3598131

[21] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2023. Adaptive REST API Testing with Reinforcement Learning. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 446–458. doi:10.1109/ASE56229.2023.00218

[22] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2025. LlamaRestTest: Effective REST API Testing with Small Language Models. doi:10.48550/arXiv.2501.08598 arXiv:2501.08598 [cs].

[23] Myeongsoo Kim, Tyler Stennett, Dhruv Shah, Saurabh Sinha, and Alessandro Orso. 2024. Leveraging Large Language Models to Improve REST API Testing. doi:10.48550/arXiv.2312.00894 arXiv:2312.00894 [cs].

[24] Myeongsoo Kim, Tyler Stennett, Saurabh Sinha, and Alessandro Orso. 2025. A Multi-Agent Approach for REST API Testing with Semantic Graphs and LLM-Driven Inputs. doi:10.48550/arXiv.2411.07098 arXiv:2411.07098 [cs].

[25] languagetool 2025. LanguageTool. https://github.com/languagetool-org/languagetool

[26] Tri Le, Thien Tran, Duy Cao, Vy Le, Tien Nguyen, and Vu Nguyen. 2024. KAT: Dependency-aware Automated API Testing with Large Language Models. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 82–92. doi:10.1109/ICST60714.2024.00017 arXiv:2407.10227 [cs].

[27] Jia Li, Jiacheng Shen, Yuxin Su, and Michael R. Lyu. 2025. LLM-assisted Mutation for Whitebox API Testing. doi:10.48550/arXiv.2504.05738 arXiv:2504.05738 [cs].

[28] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. Morest: Model-based RESTful API Testing with Execution Feedback. doi:10.48550/arXiv.2204.12148 arXiv:2204.12148 [cs].

[29] Alberto Martin-Lopez, Andrea Arcuri, Sergio Segura, and Antonio Ruiz-Cortés. 2021. Black-box and white-box test case generation for RESTful APIs: Enemies or allies?. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 231–241.

[30] Alberto Martin-Lopez, Sergio Segura, Carlos Müller, and Antonio Ruiz-Cortés. 2021. Specification and automated analysis of inter-parameter dependencies in web APIs. *IEEE Transactions on Services Computing* 15, 4 (2021), 2342–2355.

[31] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. A Catalogue of Inter-parameter Dependencies in RESTful Web APIs. In *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings* (Toulouse, France). Springer-Verlag, Berlin, Heidelberg, 399–414. doi:10.1007/978-3-030-33702-5_31

[32] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: Automated Black-Box Testing of RESTful Web APIs. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, 682–685. doi:10.1145/3460319.3469082

[33] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2022. Online Testing of RESTful APIs: Promises and Challenges. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 408–420. doi:10.1145/3540250.3549144

[34] Srinivas Nidhra and Jagruthi Dondeti. 2012. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)* 2, 2 (2012), 29–50.

[35] Vikram Nitin, Shubhi Asthana, Baishakhi Ray, and Rahul Krishna. 2022. Cargo: Ai-guided dependency analysis for migrating monolithic applications to microservices architecture. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[36] openapispec 2025. OpenAPI Specification. https://spec.openapis.org/oas/latest.html

[37] OpenRouter. 2025. OpenRouter. https://openrouter.ai. [Online; accessed Nov-2025].

[38] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2025. ASTER: Natural and Multi-language Unit Test Generation with LLMs. In *ACM/IEEE International Conference on Software Engineering*.

[39] petclinic 2025. Spring PetClinic Sample Application. https://github.com/spring-projects/spring-petclinic

[40] Marty Pitt, Dilip Krishnan, and Adrian Kelly. 2020. SpringFox. https://github.com/springfox/springfox. [Online; accessed Nov-2025].

[41] restassured 2025. REST-assured. https://rest-assured.io

[42] Diptikalyan Saha, Devika Sondhi, Swagatam Haldar, and Saurabh Sinha. 2025. REST API Functional Tester. In *Proceedings of the 18th Innovations in Software Engineering Conference*. Association for Computing Machinery, Article 8, 11 pages. doi:10.1145/3717383.3717388

[43] spring 2025. Spring. https://spring.io/

[44] Spring Team. 2013. Spring PetClinic. https://github.com/spring-projects/spring-petclinic A sample Spring-based application.

[45] springdoc. 2025. springdoc-openapi. https://github.com/springdoc/springdoc-openapi. [Online; accessed Nov-2025].

[46] Tyler Stennett, Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2025. AutoRestTest: A Tool for Automated REST API Testing Using LLMs and MARL. doi:10.48550/arXiv.2501.08600 arXiv:2501.08600 [cs].

[47] stripes 2025. Stripes Framework. https://github.com/StripesFramework/stripes

[48] struts 2025. Apache Struts. https://struts.apache.org/

[49] supplementary 2025. Supplementary Material. https://github.com/aster-test-generation/saint

[50] Swagger. 2025. Swagger Core. https://github.com/swagger-api/swagger-core. [Online; accessed Nov-2025].

[51] treesitter 2025. Tree-sitter. https://tree-sitter.github.io/tree-sitter

[52] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTEST-GEN: Automated Black-Box Testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Porto, Portugal, 142–152. doi:10.1109/icst46399.2020.00024

[53] WALA. 2025. WALA. https://github.com/wala/WALA. [Online; accessed Nov-2025].

[54] Ke Zhang, Chenxi Zhang, Chong Wang, Chi Zhang, YaChen Wu, Zhenchang Xing, Yang Liu, Qingshan Li, and Xin Peng. 2025. LogiAgent: Automated Logical Testing for REST Systems with LLM-Based Multi-Agents. doi:10.48550/arXiv.2503.15079 arXiv:2503.15079 [cs].

[55] Man Zhang and Andrea Arcuri. 2021. Adaptive hypermutation for search-based system test generation: A study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–52.

[56] Man Zhang and Andrea Arcuri. 2023. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 144 (Sept. 2023), 45 pages. doi:10.1145/3597205