Industrial Practice of LLM-Based Test Case Carving and Assertion Generation (Experience Paper)

HAOZHEN YOU, Fudan University, China
ZHEN DONG*, Fudan University, China
JINGJING WANG, ByteDance, China
QIANG LI, ByteDance, China
XIN PENG, Fudan University, China

Enterprise regression testing for microservice systems is often constrained by incomplete or outdated documentation. In practice, QA engineers frequently rely on real execution traffic to reconstruct business scenarios; however, turning raw traffic into replayable regression tests with stable validation logic remains labor-intensive and error-prone.

This paper presents NL2Test, an end-to-end approach and tool that generates executable API regression tests from (i) a natural-language scenario description and (ii) a traffic capture recorded while executing the scenario. NL2Test addresses two coupled tasks: test case carving, which extracts a minimal replayable request sequence and reconstructs data dependencies so that dynamic values are bound from their responses rather than hard-coded; and assertion generation, which produces assertions aligned with business intent while avoiding non-deterministic fields and hallucinated paths. To improve reliability, NL2Test uses LLMs for semantic interpretation and constrained code synthesis, and uses deterministic algorithms for request filtering, dependency confirmation via value consistency, and assertion-path validation.

We evaluate NL2Test on 51 industrial regression scenarios extracted from a large consumer-facing Internet company. NL2Test achieves an exact-match rate of 82.4% (42/51), and produces a functionally usable draft in 98.0% (50/51) of scenarios when allowing minor post-edits. In a 9-month production deployment starting in March 2025, NL2Test generated 3,196 test cases with an overall code adoption rate of 85.4%. These results indicate that traffic-grounded generation with deterministic guardrails can substantially reduce manual effort while improving regression automation in complex microservice environments.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Test Carving, Assertion Generation, API testing

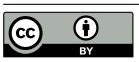
ACM Reference Format:
Haozhen You, Zhen Dong, Jingjing Wang, Qiang Li, and Xin Peng. 2026. Industrial Practice of LLM-Based Test Case Carving and Assertion Generation (Experience Paper). *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA008 (October 2026), 23 pages. https://doi.org/10.1145/3832099

1 Introduction

Enterprise software is increasingly organized as large-scale ecosystems of microservices, where requirements and APIs evolve rapidly and releases are frequent. In such settings, regression testing

*Corresponding author.

Authors' Contact Information: Haozhen You, Fudan University, Shanghai, China, hzyou23@m.fudan.edu.cn; Zhen Dong, Fudan University, Shanghai, China, zhendong@fudan.edu.cn; Jingjing Wang, ByteDance, Shanghai, China, wangjingjing.vector@bytedance.com; Qiang Li, ByteDance, Shanghai, China, liqiang.leo@bytedance.com; Xin Peng, Fudan University, Shanghai, China, pengxin@fudan.edu.cn.



ACM 2994-970X/2026/10-ARTISSTA008
https://doi.org/10.1145/3832099

of cross-service business flows remains a substantial engineering burden. Although many testing activities have been automated, API regression tests for these flows are still largely constructed and maintained by QA engineers. Given a natural-language (NL) scenario description, engineers need to identify the relevant service endpoints, recover a replayable call sequence, bind dynamic values such as IDs and tokens across steps, and encode business expectations as executable checks [4, 29]. This process is labor-intensive, error-prone, and difficult to scale: each API change or workflow update may require manual inspection of traces or logs, repair of data dependencies, and revision of assertions. As a result, generating and maintaining regression tests has become a practical bottleneck in industrial microservice systems.

Existing test-generation techniques provide only limited relief in this setting [13, 15, 28]. Many techniques target a single module, class, or service, whereas real business scenarios often span multiple services. Others derive expected values from current executions, which can tie generated assertions to transient system behavior rather than intended business outcomes. Many approaches also assume isolated, stable, and resettable environments, an assumption that does not hold for many online services with shared state, dynamic content, and production-like dependencies [8, 32, 34]. Consequently, generated tests may be difficult to replay reliably, brittle under data changes, or insufficient for checking the intended business behavior [17, 21].

LLMs appear promising for reducing this manual burden because they can interpret natural-language scenarios and synthesize executable code. However, applying them directly to industrial microservice regression testing is difficult. The first challenge is selecting the business-relevant API sequence from noisy raw traffic. In practice, a captured trace may include background polling, authentication, logging, retries, monitoring requests, and calls from unrelated UI components. An LLM must distinguish the key APIs of the target business flow, preserve their order, and retain the dynamic data dependencies needed for replay. The second challenge is turning abstract business expectations into executable assertions. NL scenarios often describe outcomes such as successful creation, approval, or rule triggering, but do not specify which API response to inspect, which field or path to check, which operator to use, or which values are stable across executions. Consequently, an unconstrained single-pass NL-to-test approach may hallucinate endpoints, parameters, response fields, or assertions, leading to tests that are executable but irrelevant, flaky, or semantically incorrect [7, 35].

This paper presents NL2Test, an end-to-end LLM-based tool deployed in an industrial microservice testing environment to assist QA engineers in generating API regression tests. Given a natural-language scenario description and raw traffic captured during scenario execution, NL2Test produces an executable test draft for engineer review and integration into existing regression suites. Rather than treating test generation as a single-step natural-language-to-code task, the paper reports the design choices required to make this agent-based workflow practical in this setting: carving the target business flow from noisy traffic, reconstructing dynamic data dependencies such as IDs and tokens for replay, and translating abstract natural-language expectations into concrete test assertions. A central focus is where LLMs should be used for semantic interpretation and where deterministic mechanisms are needed to validate generated artifacts and mitigate hallucinations, such as irrelevant API selection, invalid assertion paths, and incorrect variable references.

As an industrial experience report, this paper contributes lessons and evidence from building, deploying, and evaluating NL2Test. First, we characterize the practical obstacles that arise when applying LLM-assisted test generation to production microservice workflows, including under-specified NL descriptions, outdated documentation, noisy traffic captures, dynamic cross-request data dependencies, and unstable execution environments. Second, we present the traffic-grounded design of NL2Test, in which noisy traces are carved into replayable API sequences with runtime data bindings, and response-grounded assertion generation maps business intent to executable

checks while filtering invalid paths and non-existent fields [22]. Third, we report empirical results from both offline evaluation and deployment. On 51 industrial regression scenarios, NL2Test achieves an exact-match rate of 82.4% and produces usable drafts for 50 scenarios. During the first nine months after its deployment in March 2025, NL2Test generated 3,196 regression tests, of which 2,730 were adopted by engineers, yielding an adoption rate of 85.4%. These findings suggest that traffic-grounded LLM assistance can be integrated into production QA workflows, while also revealing limitations and lessons on where semantic LLM reasoning should be combined with deterministic validation.

2 Traffic-Grounded Regression Test Generation in Industry

2.1 Scenarios, Traffic Captures, and Target Tests

In enterprise microservice systems, a user-visible business workflow is often implemented by a sequence of API calls across multiple services. In practice, however, engineers may not have access to complete source code, stable endpoint documentation, or up-to-date API specifications. When constructing regression tests for an existing business scenario, they often rely on two artifacts that are readily available in industrial testing workflows: a natural-language scenario description and a traffic capture recorded during one execution of that scenario.

These two artifacts provide complementary information. The natural-language scenario description specifies the business intent, user actions, and expected outcome. For example, it may describe that a user favorites a hashtag and then expects the hashtag to appear in the user's profile hashtag list. However, such a description does not specify the concrete backend APIs, request parameters, response fields, or data dependencies needed to implement the test. The traffic capture, in contrast, records the concrete HTTP/HTTPS requests and responses produced during the scenario execution. In Web systems, HAR is a common representation because it preserves request paths, methods, headers, cookies, request and response payloads, and timing information in a structured format [24, 26]. Therefore, traffic captures provide execution evidence that can ground API-level regression-test generation.

However, a traffic capture is not itself a regression test. It usually over-approximates the target scenario. A short user operation may trigger many requests, including static resource loading, background polling, preflight checks, retries, logging, recommendation refreshes, and unrelated UI-triggered calls. Replaying the entire capture would produce long and brittle tests that are sensitive to irrelevant implementation details and nondeterministic background behavior. Conversely, relying only on the natural-language scenario is insufficient because it lacks executable API-level details. The key problem is therefore to combine the semantic guidance from the scenario with the execution evidence from the traffic capture.

The target output in this paper is an executable API regression test for the given scenario. Such a test should satisfy three requirements. First, it should contain a compact sequence of business-relevant API calls carved from the noisy traffic. Second, it should be replayable across executions: dynamic values such as object identifiers, cursors, tokens, and session-dependent parameters should be extracted from earlier responses and reused in later requests, rather than being blindly hard-coded from the recorded trace. Third, it should contain business-level assertions over concrete response fields, so that the generated test checks whether the user-visible expectation in the scenario is satisfied instead of merely checking HTTP status codes.

In current industrial practice, engineers perform this process manually [3, 33]. They inspect the captured traffic, identify requests that correspond to scenario steps, remove irrelevant calls, recover cross-request data dependencies, and translate natural-language expectations into executable assertions. This process is feasible for simple traces, but becomes labor-intensive and error-prone

in large microservice systems where a single user action may trigger many auxiliary requests and where key runtime values are generated dynamically. This motivates traffic-grounded regression-test generation: given a natural-language scenario description and a recorded traffic capture, automatically produce a compact, replayable, and assertion-rich API regression test [9, 12, 14].

2.2 Running Example: Favorite Hashtag

Fig. 1 shows a representative production scenario used as a running example in this paper. The scenario, titled *Favorite Hashtag*, describes a common interaction in a content-consumption application: a user clicks a hashtag in the For You feed to navigate to its detail page, adds the hashtag to Favorites, and then checks whether the newly favorited hashtag appears in the user's Profile Hashtags list.

1. **[Action] Click on a hashtag to navigate to its detail page.**
2. **[Action] Click to favorite the hashtag. [Expect] The hashtag is successfully favorited.**
3. **[Action] Navigate to the hashtag list on the personal page. [Expect] The list contains the newly favorited hashtag.**

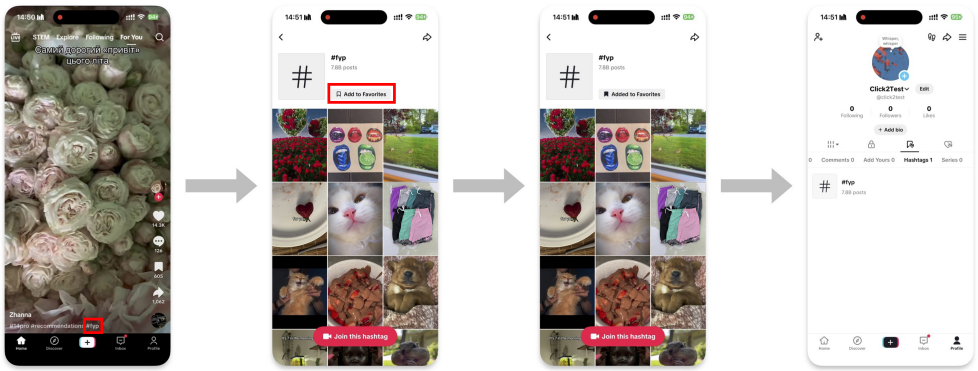


Fig. 1. Example of traffic-grounded test carving and assertion generation.

Although the scenario is concise from the user's perspective, the recorded traffic is much larger and noisier. Besides the requests that implement the hashtag-favoriting workflow, the capture may also include requests for resource loading, recommendation updates, logging, background refreshes, and other auxiliary interactions. A generated regression test should therefore not replay the whole capture. Instead, it should carve out the key API calls that realize the scenario, such as retrieving the selected hashtag from the feed or hashtag-detail response, sending the request that favorites the hashtag, and querying the Profile Hashtags list to validate the final state.

The carved sequence must also be made replayable. For example, the hashtag identifier observed in the recorded execution should not be hard-coded into the generated script. The test should extract the identifier, such as `hashtag_id`, from an earlier response and reuse it in the later favorite request and in the final Profile Hashtags assertion. Without such dependency reconstruction, the generated script may pass only for the recorded execution and fail when runtime data changes in later executions.

Finally, the test needs business-level assertions. In this example, the generated assertions should check that the relevant feed or detail request succeeds, that the response contains a valid hashtag, that the favorite operation succeeds and marks the target hashtag as favorited, that the Profile

Hashtags request succeeds, and that the returned Profile Hashtags list contains the same hashtag selected and favorited in the previous steps. These checks connect the natural-language expectation in the scenario to concrete fields in real API responses.

This running example illustrates the end-to-end task addressed in this paper. Given a natural-language scenario and a noisy traffic capture, the system must identify business-relevant requests, reconstruct data dependencies across requests, and generate executable assertions over response fields. NL2Test is designed for this setting: it performs traffic-grounded test carving and assertion generation to produce replayable API regression tests from industrial traffic captures.

3 LLM-Based Regression Test Generation for Microservice Systems

Our approach, NL2Test, generates regression tests for microservice systems from a natural-language business scenario and a traffic trace recorded during a reference execution. To handle noisy traces, runtime-dependent values, and meaningful test oracles, the agent-based workflow is organized into two stages: *Test Script Carving* and *Assertion Generation*. The first stage reconstructs a replayable business workflow by filtering irrelevant requests, matching scenario steps to request records, parameterizing runtime-dependent values such as tokens and object identifiers, and assembling the executable script. The second stage turns the replayable script into a regression test by deriving expected outcomes from the scenario, mapping them to concrete response fields, validating these fields against recorded responses, and inserting executable assertions into the script.

3.1 Test Script Carving

Test script carving transforms a recorded traffic trace into a compact and executable API workflow. In our experience, traffic collected from microservice systems cannot be directly replayed as regression tests: it contains infrastructure requests, background service calls, and API calls unrelated to the target business scenario. Therefore, this stage extracts only the business-relevant trace and turns it into a runnable test skeleton. As shown in Fig. 2, NL2Test implements this stage as an agent-based workflow consisting of four coordinated steps: traffic filtering, LLM-based semantic alignment, dynamic data parameterization, and script assembly.

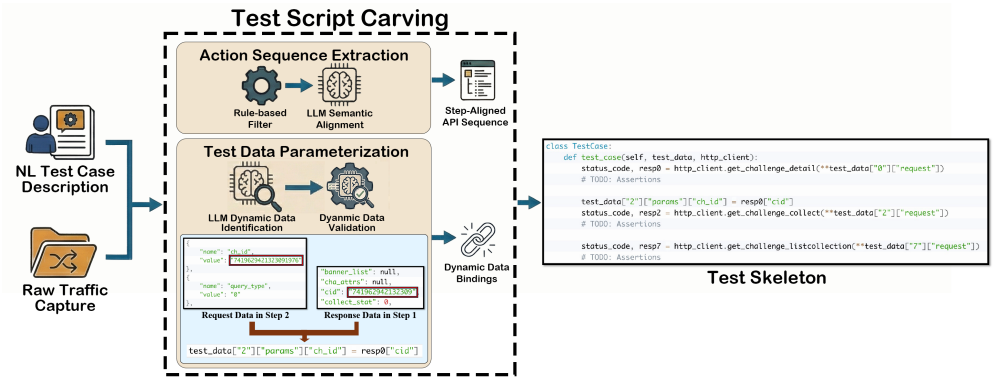


Fig. 2. Overview of Test Script Carving.

3.1.1 Traffic Filtering. We first reduce the raw traffic through rule-based filtering before invoking the LLM. A practical lesson from our experience is that many irrelevant requests can be identified without semantic reasoning. We mainly filter the following categories of requests:

- **Static resource requests.** These include JavaScript files, CSS files, images, fonts, icons, and other assets used for page rendering. They can usually be identified by path suffixes, content types, or cache-related headers.
- **Protocol-level requests.** These include browser- or runtime-generated requests such as CORS preflight `OPTIONS` calls. They are required for protocol negotiation, but do not represent user-level business actions.
- **Infrastructure and periodic requests.** These include heartbeat, keep-alive, logging, prefetching, background polling, and similar requests. They are often triggered by the application runtime rather than by the scenario step itself.

Sending all these requests to the LLM increases token cost and may distract the semantic-alignment agent from the API calls that actually implement the business workflow, making LLM-based semantic alignment less stable.

Our approach therefore applies conservative rule-based filtering based on request-level signals, including HTTP method, path suffix, content type, and repeated polling patterns. The filtering is intentionally conservative: if a request cannot be confidently classified as noise, it is kept for later semantic matching. This design choice reflects a trade-off we observed in practice. Removing a business-relevant request is difficult to recover from, whereas keeping a few extra candidates only increases the burden of the subsequent matching step.

Rationale. Rule-based filtering is effective for removing structurally obvious noise. Within the agent-based workflow, this deterministic preprocessing step reduces the search space before semantic reasoning is invoked. LLMs are more useful after this reduction step, where the remaining task requires semantic understanding rather than simple pattern recognition.

3.1.2 LLM-based Semantic Alignment. After filtering, NL2Test uses a semantic-alignment agent powered by an LLM to perform semantic alignment between natural-language scenario steps and candidate endpoint records. Instead of asking the LLM to directly generate test code, we formulate this stage as a constrained prompt-based alignment task. Given the scenario description and the filtered traffic, the agent first infers the likely purpose of each request from its endpoint path, request parameters, and response fields, and then maps scenario steps to the corresponding request records.

This intermediate step-to-endpoint mapping is important in practice. In microservice traces, different requests may have similar paths or parameter names but correspond to different business actions. Directly generating a script from the scenario and traffic made such mistakes hard to inspect. By requiring the semantic-alignment agent to output an explicit step-to-endpoint mapping, engineers can check whether the extracted workflow is correct before the script is assembled.

The alignment prompt imposes several constraints. Each scenario step can be aligned with at most one endpoint record, and each endpoint record can be used for at most one step. The aligned endpoint records must preserve their chronological order in the original trace. The agent is also allowed to leave a step unmatched, since some scenario descriptions describe UI observations or navigation actions that do not trigger business API calls. Prompt 1 shows the prompt used in this stage. Its output includes the request method, endpoint path, request identifier, and inferred request purpose, which together serve as the basis for script generation.

We further use few-shot examples to stabilize both the alignment pattern and the output format. The examples are selected from manually verified development cases and are excluded from the evaluation benchmark. Rather than selecting examples by business domain, we choose them by decision pattern: straightforward step-to-endpoint alignment, steps with no corresponding endpoint, and cases where temporal order is needed to distinguish similar endpoints. This strategy

helps the agent follow the alignment constraints and produce structured output without overfitting to product-specific endpoint names or business objects.

Prompt 1: Prompt for Step Alignment

Input Data

Test Steps: `{{TEST_DESC}}`

Filtered Traffic Data: `{{REQ_LIST}}` .

Task List

Task 1: Infer the likely purpose of each candidate endpoint from its path and request/response content.
Task 2: Match each step to at most one endpoint, and each endpoint to at most one step.

Constraint

1. Preserve the original step order; Do not match a later step to an earlier request.
2. Do not force a match when no business endpoint is triggered.

Output: A JSON object containing step–endpoint mappings and endpoint purposes.

Few-shot Chain-of-Thought Example

(Same overall structure as above, not shown extensively here)

Rationale. Asking the agent to produce an inspectable intermediate representation is more reliable than asking it to generate test code directly. The step-to-endpoint mapping makes semantic errors visible early and allows deterministic procedures in subsequent workflow steps to assemble the executable script.

3.1.3 Dynamic Data Parameterization. A correct endpoint sequence is still insufficient for regression testing. In practice, many generated scripts fail because they contain captured runtime values, such as user IDs, object IDs, content IDs, or session-dependent tokens. These values may be valid in the recorded execution but invalid when the test is replayed later. NL2Test addresses this problem through dynamic data parameterization, whose goal is to replace hard-coded request parameters with variables extracted from earlier responses.

NL2Test decomposes dynamic data parameterization into *LLM Dynamic Data Identification* and *Dynamic Data Validation*. In the agent-based workflow, the former acts as a semantic identification agent, while the latter combines deterministic validation with an LLM-based semantic check. This is motivated by the observation: directly asking the LLM to infer request dependencies is unreliable, because LLM may rely on field-name similarity and invent plausible but incorrect links. Instead, NL2Test first uses the LLM only to identify candidate dynamic fields, and then validates their dependencies using evidence from the recorded trace.

First, NL2Test uses the dynamic-data identification agent to identify request parameters that are likely to represent key business resources in the workflow. This step is semantic: the same business object may appear under different endpoint names or parameter conventions, and a single request may contain both control parameters and business identifiers. Thus, NL2Test uses this agent only to narrow the search space to resource-related request fields, rather than to directly infer dependencies.

Second, NL2Test validates whether the identified dynamic fields can be safely replaced by values extracted from previous responses. For a candidate parameter in request r_i, the system retrieves its recorded value and searches previous responses $r_j.response$, where $j < i$, for the same value. If the value appears in a previous response, NL2Test records a candidate dependency from the response field to the later request parameter. This value-consistency check grounds the dependency in the observed trace and avoids relying only on field-name similarity.

Value consistency alone, however, can still be misleading, because the same value may appear in unrelated response fields. Therefore, NL2Test further applies a consistency check during Dynamic Data Validation. Given the source response path and the target request parameter path, the agent determines whether the two fields refer to the same business resource. A dependency is accepted only when both value consistency and semantic consistency hold. In cases with multiple valid candidates, NL2Test selects the earliest source that satisfies the semantic consistency check, following the chronological order of the recorded workflow.

The confirmed dependencies are used to rewrite the script. Hard-coded request values are replaced by variable references extracted from earlier responses. As a result, the generated test no longer depends on the concrete IDs observed in the original capture.

Rationale. NL2Test uses LLM-powered agents only where semantic judgment is needed, while relying on value consistency to ground dependencies in the recorded trace.

3.1.4 Script Assembly. Given the step-aligned endpoint sequence and the reconstructed dependency set, NL2Test completes the carving workflow by invoking a deterministic script-assembly step to assemble the executable test skeleton using templates. The templates encode stable script structure, including request construction, response parsing, variable extraction, and request sequencing. This avoids asking the LLM agents to generate the full script from scratch, reducing formatting errors and structural drift.

The resulting test skeleton contains the business-relevant API sequence and the necessary cross-step data flow. Assertion code is not generated at this stage; instead, NL2Test leaves explicit insertion points for the later assertion generation stage.

3.2 Assertion Generation

After test script extraction, NL2Test obtains a replayable API workflow, but replayability alone does not make it a regression test. The workflow must also check whether the system behavior is correct. This is challenging because recorded responses contain many business-irrelevant fields, such as timestamps, trace identifiers, pagination cursors, and diagnostic messages, and treating them as expected outputs leads to brittle snapshot-style tests.

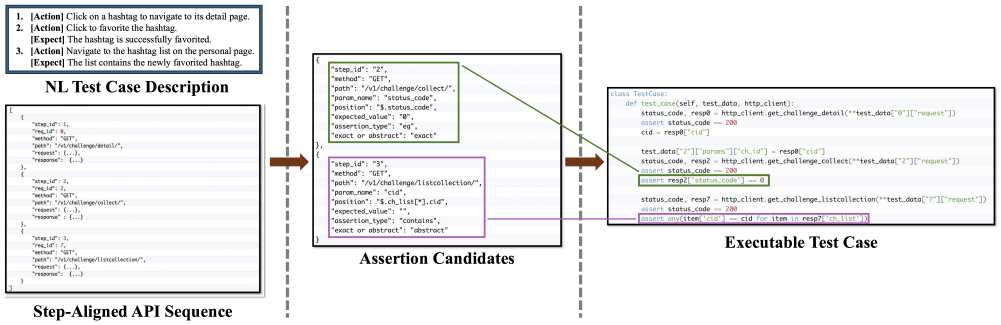


Fig. 3. Overview of assertion generation and deterministic validation.

Fig. 3 illustrates how NL2Test uses structured assertion candidates as an intermediate representation between the input scenario/traffic and the final executable test case. Each candidate records the target step, endpoint, response field, expected value, and assertion type, and is later translated into assertion code. Internally, NL2Test obtains these candidates through assertion intent extraction, deterministic assertion validation, and constrained assertion code generation. This design assigns

bounded semantic decisions to LLM-powered agents, while leaving field grounding and script construction to deterministic procedures.

3.2.1 Assertion Intent Extraction. NL2Test first identifies what should be verified after each business step. Instead of directly generating assertion code, an assertion-intent agent outputs structured **assertion candidates**. Each candidate records the step identifier, the matched endpoint, the target response field and its path, the assertion type, the expected value, and the expected-value form, i.e., *exact* for fixed hard-coded values or *abstract* for values resolved from dynamic variables during test execution.

Prompt 2 takes as input the natural-language test description, the step-aligned endpoint sequence, and compact response summaries. It asks the agent to infer business expectations from each scenario step and map them to observable fields in the corresponding endpoint response. This design avoids asking the agent to assert arbitrary fields from the whole trace.

Prompt 2: Assertion Intent Extraction

Input Data

Test Step: `{{TEST_DESC}}`

Request Sequence Set after Semantic Extraction: `{{REQ_SEQ}}` .

Task: Identify the business expectations implied by each step. Map each expectation to one or more response fields from the corresponding step.

Constraint

1. If the step contains `[expect]`, all mentioned fields must appear in the final assertion.
2. For state-change operations (e.g., like, follow, collect, upload, delete, update), verify that data was correctly added, removed, or updated. For list structures, use inclusion checks.
3. For every operation that succeeds, if a status code or equivalent field exists in the schema, a status code equality assertion must be generated.
4. Do not use runtime-only fields such as timestamps or log identifiers.

Output: A JSON array of assertion descriptors.

Few-shot Chain-of-Thought Example

(Same overall structure as above, not shown extensively here)

The prompt imposes several constraints. Explicit `[expect]` statements should be covered when they are observable from the response. For state-changing operations, such as like, follow, collect, upload, delete, or update, the candidate should check whether the relevant data has been added, removed, or updated. For list responses, containment-style assertions are preferred over full-list equality. Runtime-only or infrastructure fields, such as timestamps, trace identifiers, log identifiers, request durations, and diagnostic messages, are excluded. The agent may also output no Assertion Candidate for a step if no stable business-observable outcome is exposed.

We use few-shot examples to stabilize both the candidate format and the oracle-selection logic. The examples are selected from manually reviewed development cases and excluded from the evaluation benchmark. They are chosen by oracle pattern rather than by product domain, covering cases such as status-code equality, state-change confirmation, list containment, and abstract expected values.

The resulting assertion candidates separate semantic oracle inference from executable code generation. The assertion-intent agent proposes what should be checked, while later stages validate whether the fields exist, whether the expected values are stable, and how the assertions should be encoded.

Rationale. Structured assertion candidates make the agent's oracle decisions inspectable and allow invalid or unstable checks to be filtered before code generation.

3.2.2 Deterministic Assertion Validation. The assertion candidates produced by the assertion-intent agent may still contain invalid field paths, ambiguous field references, or unstable expected values. Therefore, NL2Test validates each Assertion Candidate against the recorded response before generating code. This step does not introduce new assertion intent; it only repairs, normalizes, or removes candidates proposed by the agent.

The two highlighted candidates in Fig. 3 show how this validation works in practice. In the first case, the agent proposes an *exact* assertion over the response-body field `$.status_code` of the collect request. NL2Test checks that this path exists in the recorded response and retains the stable expected value `0`. The generated test therefore contains a business-level assertion over `resp2['status_code']`, in addition to the generic HTTP-level assertion `status_code == 200`. This distinction is important because the HTTP status code only indicates transport-level success, while the response-body status field captures the application-level outcome.

In the second case, the agent proposes an *abstract* containment assertion over `$.ch_list[*].cid` for the list-collection request. NL2Test checks that the list field and its element field exist in the recorded response. Since the expected value is marked as abstract, the validator removes any hard-coded identifier produced by the agent and binds the assertion to the runtime variable `cid` obtained from an earlier step. Code generation then emits a containment check over the returned list, rather than a full-list equality assertion or a brittle hard-coded ID comparison.

For each Assertion Candidate, NL2Test validates the target field and expected-value form before code generation. It first checks whether the referenced response path exists. If the path is invalid, NL2Test repairs it by field-name search only when the response contains a unique matching field; otherwise, the candidate is discarded.

NL2Test then normalizes the expected value. Exact candidates are grounded to stable constants from the recorded response, whereas abstract candidates are treated as runtime-dependent values and are not hard-coded. Stable status fields, such as `status`, `status_msg`, and `error_msg`, are translated into exact equality assertions. Invalid or ambiguous candidates are removed to avoid brittle tests.

This step deliberately favors executable and stable assertions over aggressive assertion coverage. Removing an invalid Assertion Candidate may reduce the number of checks, but keeping a candidate that points to a non-existent field or hard-codes a dynamic value would generate a test that fails for the wrong reason.

Rationale. Deterministic validation grounds the agent-generated candidates in recorded evidence, ensuring that every generated assertion references an actual response field and uses the correct expected-value form before code generation.

3.2.3 Constrained Assertion Code Generation. After validation, NL2Test converts the refined assertion candidates into executable assertion statements and inserts them into the test skeleton. The code-generation agent's in this step is deliberately narrow. It is not allowed to invent new assertion targets, helper functions, response fields, or business logic. Instead, it receives the validated assertion descriptors, the relevant response structures, and the existing test skeleton, and generates code only for predefined insertion points.

This constrained generation is useful for assertions beyond simple templates, such as nested JSON checks, list-containment checks, and comparisons against variables extracted earlier in the workflow. Since assertion targets and expected-value forms have been validated, the agent only adapts the assertion code to the surrounding script context, including variable scope, indentation, response-access patterns, and the assertion library.

The final output is a complete executable regression test that combines the parameterized API workflow with the generated assertions. It can be replayed without invoking the agent-based workflow again. Regeneration is needed only when the business workflow, endpoint contract, response schema, or assertion semantics change materially.
Rationale. The code-generation agent acts only as a bounded code translator: validated assertion candidates decide what to check, while the agent expresses those checks in the existing test skeleton.

4 Evaluation

4.1 Benchmark Construction

To evaluate the generalization capability of NL2Test in a real-world enterprise environment[31], we constructed a comprehensive benchmark dataset containing **51 distinct regression scenarios** collected from the production branches of five series at a large consumer-facing Internet company. Although all scenarios come from the same enterprise setting, these five series differ substantially in business style and technical complexity. This is because NL2Test requires natural-language descriptions of problematic test cases together with the corresponding server-side traffic, which are proprietary and generally inaccessible from other companies. As a result, we cannot conduct experiments across more enterprises or external environments, and instead evaluate NL2Test across diverse business lines within our own enterprise environment. The benchmark covers a range of application behaviors, from content consumption to content creation and transactional workflows. To maintain the conciseness of the main text, the detailed list of these scenarios and their specific attributes are provided in the Appendix A. The benchmark covers the following domain-specific categories:

- **C**-Series: Scenarios often require multiple assertions per step to check state changes and data integrity.
- **U**-Series: These scenarios tend to have higher noise in the recorded traffic due to repetitive user actions.
- **I**-Series: The interactions in these scenarios are more complex, often involving dynamic media and UI-related data manipulations .
- **M**-Series: These scenarios include workflows with dynamic parameter binding between steps.
- **S**-Series: Scenarios in this series focus on accuracy in handling dynamic queries, often dealing with complex check logic.

These scenarios are challenging for both carving and assertion generation. In production traffic, only a small fraction of captured requests corresponds to the actual business workflow. For example, in the representative *U2* scenario, the system must identify only 2 valid business APIs from more than 300 raw requests. In addition, many scenarios require more than simple status checks and instead involve multiple business assertions over state changes or returned data.

4.1.1 Ground Truth Formulation. For each scenario, we curated a triplet consisting of a natural-language test description, a raw traffic capture, and a ground-truth script. **The ground-truth scripts were manually written and reviewed by senior QA engineers** to ensure that they correctly reflect the intended business logic and handle cross-step dynamic dependencies. We use these scripts as the reference for evaluating the functional correctness of generated tests.

4.1.2 Model Selection and Setup. For the experimental evaluation, we used a high-performance reasoning-oriented LLM, referred to as *Enterprise-LLM* due to non-disclosure constraints on the underlying infrastructure. The model was deployed in a private cloud environment to ensure enterprise data isolation during generation.

Importantly, NL2Test is an independent generation module rather than a system coupled to enterprise-specific infrastructure: except for benchmark construction and deployment on internal servers, its pipeline depends only on scenario descriptions, captured traffic, and LLM calls, making the reported effectiveness attributable to the generation method itself.

4.2 RQ1: Effectiveness and Efficiency of Generation

To answer whether NL2Test can generate functionally equivalent test scripts, we define a unified evaluation metric based on the generated code's execution and assertion status. An **Exact Match** case is one where the API sequence, dynamic parameter passing, and assertion logic are all strictly consistent with the ground truth, denoted by a checkmark (✓). Cases that correctly identify the business workflow sequence but contain defects—such as missing or mismatched parameter dependencies or incomplete assertions—are classified as **Partial Match**, marked with an asterisk (*). Any case failing to identify the correct API sequence is deemed Incorrect.

In the detailed results table below, the headers provide specific metrics for each scenario. *Origin / Valid* contrasts the total number of raw captured requests against the number of valid business APIs, highlighting the noise level. *Asserts* indicates the assertion complexity by counting the number of business rules in the ground truth. *Defects (S/P/A)* reports the number of missing or incorrect workflow steps, parameter dependencies, and assertions, respectively. *Tokens* and *Time* quantify the computational cost in terms of LLM usage and total generation latency. *Outcome* shows the final classification result.

Table 1. Detailed Evaluation Metrics including Accuracy, Token Usage, Execution Time, and Defect Counts.

ID	Description Title	Origin / Valid	Asserts	Tokens (Prompt / Compl.)	Time(s)	Defects (S/P/A)	Outcome
C1	Discover favorite hashtag	8 / 3	4	25,457 / 12,095	442	/	✓
C2	View private profile videos	3 / 3	4	49,094 / 5,491	204	/	✓
C3	Unfavorite video on profile	3 / 2	3	16,654 / 10,169	349	(0/1/0)	✓*
C4	Favorite video to new collection	4 / 3	4	17,245 / 8,198	289	/	✓
C5	Create new video collection	9 / 4	6	45,772 / 10,935	352	/	✓
C6	Delete video collection	4 / 3	3	16,547 / 7,100	242	/	✓
C7	Cancel favorite hashtag	8 / 3	5	23,646 / 7,485	230	/	✓
C8	View posts list video	3 / 2	3	38,537 / 5,735	198	/	✓
C9	Set feed recommendation content	4 / 3	4	18,179 / 6,888	228	(0/0/1)	✓*
C10	Delete feed history instruction	3 / 2	3	38,609 / 5,687	204	/	✓
C11	Delete feed instruction (Long press)	4 / 2	3	16,649 / 4,139	143	/	✓
C12	Set new feed preference	4 / 4	6	35,429 / 10,036	343	/	✓
C13	Unfavorite hashtag from list	5 / 3	3	38,863 / 5,221	173	/	✓
C14	Update profile pronouns	3 / 2	3	19,993 / 7,437	243	/	✓
C15	Switch to private account	3 / 1	2	18,581 / 6,920	238	/	✓
C16	Switch to Business Account	5 / 1	1	17,811 / 5,872	212	/	✓
C17	Reorder profile advanced features	3 / 2	2	16,683 / 7,323	253	/	✓
C18	Link account	4 / 1	2	16,829 / 4,222	144	(0/0/1)	✓*
U1	Like video (Logged-in)	157 / 3	6	104,341 / 9,980	353	/	✓
U2	Login popup (Logged-out)	307 / 2	2	97,643 / 6,908	259	(1/0/2)	
U3	Follow another user	152 / 2	4	29,298 / 5,332	194	(0/1/0)	✓*
U4	Favorite video	62 / 2	4	71,002 / 7,208	273	/	✓
I1	Select video template	2 / 2	4	16,358 / 3,700	132	/	✓
I2	View video effect details	2 / 1	2	15,728 / 4,271	154	/	✓
M1	Artist: Upload & delete song	113 / 1	2	82,356 / 10,572	372	/	✓
M2	Artist: Upload & view releases	77 / 2	5	39,066 / 6,587	245	(0/0/2)	✓*
M3	Edit and save release draft	165 / 2	3	108,567 / 7,689	311	/	✓
M4	Delete draft from 'My Releases'	58 / 3	5	40,115 / 8,049	314	/	✓
M5	Download & view AOP contract	39 / 1	2	19,638 / 5,450	206	/	✓
M6	View artist insights	51 / 2	4	36,860 / 4,613	184	/	✓
M7	Modify 'Under Review' song title	87 / 3	4	59,782 / 8,649	618	/	✓
M8	View artist 7-day trend data	13 / 2	3	20,561 / 9,718	333	/	✓
M9	Sort artist songs by posts	10 / 3	6	21,437 / 6,197	215	/	✓

ID	Description Title	Origin / Valid	Asserts	Tokens (Prompt / Compl.)	Time(s)	Defects (S/P/A)	Outcome
M10	Search and star artist	11 / 3	4	19,900 / 6,826	238	/	✓
M11	Artist: Login & view songs	144 / 3	5	40,624 / 10,877	367	/	✓
M12	Artist: Modify 'Under Review' song	163 / 4	7	69,856 / 6,673	235	/	✓
M13	Artist: Delete rejected song	23 / 2	2	23,537 / 6,087	220	/	✓
M14	Label: Save draft & delete	158 / 3	5	46,081 / 7,419	261	/	✓
M15	Label: Upload artist avatar	58 / 3	5	32,800 / 8,104	603	/	✓
M16	Label: Add new artist	60 / 4	7	36,356 / 8,131	269	(0/0/1)	✓*
M17	Exit song edit without save	95 / 2	4	59,497 / 7,008	229	/	✓
M18	Artist: Login & change avatar	162 / 1	1	48,791 / 5,780	208	/	✓
M19	Creator joins WWA	4 / 1	2	14,219 / 4,158	147	/	✓
M20	Creator submits WWA video	18 / 3	5	26,134 / 11,603	392	(0/0/1)	✓*
M21	Creator submits self-serve video	10 / 1	2	16,382 / 4,286	164	/	✓
M22	View joined WWA campaigns	14 / 2	3	20,025 / 5,328	202	/	✓
M23	Favorite WWA campaign	16 / 3	5	26,269 / 7,859	280	/	✓
M24	View WWA bills	14 / 2	3	19,853 / 8,443	312	/	✓
M25	Subscribe to WWA campaign	14 / 2	3	20,391 / 5,630	203	/	✓
S1	Unit conversion search	1 / 1	4	36,335 / 8,700	308	/	✓
S2	Game card search	1 / 1	3	16,739 / 5,886	222	(0/0/1)	✓*

4.2.1 Effectiveness Analysis. The evaluation demonstrates that NL2Test is highly robust in producing production-ready scripts, achieving a **Total Correct Rate of 98.0% (50/51)**, with **82.4% (42/51) of these being Exact Match**. The high "Exact Match" rate is attributed to two key architectural choices tailored to different complexities. First, for scenarios with massive traffic volume (e.g., U1 with 157 requests), our Hybrid Denoising strategy—which rigorously filters structural noise via rules before applying the semantic-alignment agent—proved critical in isolating the valid API signal. Second, for scenarios requiring complex assertion (e.g., M12 with 7 assertions), the success is driven by our structured agent-based assertion generation workflow. By explicitly separating the identification of assertion entities from the analysis of their concrete values, the system can systematically construct multi-point assertions that match human-level rigor.

Regarding the "Partial Match" cases (15.6%), the primary failure mode stems from the gap between text descriptions and human intuition. In Case S2 (Game Search), the text simply stated "Search input: Dota2" without defining specific checks. Consequently, the assertion-intent agent generated a generic assertion checking that the result list was non-empty, while the human ground truth additionally checked that the result title contained "Dota2". This discrepancy highlights that while the agent followed the provided instruction, it lacked the implicit domain habit of checking content relevance unless explicitly prompted. Similar assertion-side defects also appear in C9, C18, M2, M16, and M20. These cases represent valid but under-asserted scripts that are easy to patch. A secondary, less frequent cause is unmatched parameter names due to non-standard API definitions. In Case U3, the dynamic-data identification and validation steps failed to link a dependency because the API returned an ID with the obscure name 'odidId' instead of a standard name like 'userId'. Because the names looked completely unrelated, the agent cautiously avoided linking them.

The single Incorrect case (U2) reveals the tool's limitations when facing Repetitive Patterns combined with extreme noise. In this scenario, the user repeatedly invoked the same endpoint with varying parameters amidst 300+ background requests. The combination of an Extreme Signal-to-Noise Ratio and the similarity of repeated requests confused the semantic-alignment agent's step alignment, causing it to fail in identifying the correct sequence. The defect-count column further confirms that this failure is not merely an assertion-level issue, but involves an incorrect workflow step together with assertion omissions.

4.2.2 Efficiency Analysis. In addition to effectiveness, we also analyze the efficiency of NL2Test in terms of generation latency and LLM token usage. Across all 51 scenarios, NL2Test takes an

average of **265 seconds** to generate a test script. The average token consumption is **35.4k input tokens** and **7.2k output tokens** per case, resulting in an average total of **42.6k tokens**. Reporting input and output tokens separately provides a clearer view of the computational cost: the input side mainly reflects the cost of processing captured traffic, intermediate analysis results, and task descriptions, while the output side reflects the generated scripts and reasoning artifacts.

The token consumption varies substantially with the size of the captured traffic. For example, U1 and M3 require over 100k input tokens because they contain 157 and 165 raw requests, respectively, and the system needs to process large capture files during agent-based semantic alignment and workflow reconstruction. Execution time is also affected by the complexity of iterative assertion refinement, as scenarios with more complicated business logic or assertion construction may require additional interactions among the assertion-intent and code-generation agents.

Although an average latency of around four minutes is longer than instantaneous code completion, it remains practical in the context of integration-test generation. Manually writing, debugging, and validating a complex integration test often requires substantial engineering effort. Therefore, trading a few minutes of machine generation time for an executable test script can still provide meaningful productivity gains, especially for workflows involving noisy traffic, multi-step API dependencies, and non-trivial assertions.

4.3 RQ2: Maintainability and Practical Quality in Deployment

To evaluate the long-term maintainability and utility of the generated scripts, we introduced NL2Test into the deployment environment at the same company starting in **March 2025**. By the data cutoff on December 2025, the tool had successfully generated a total of **3,196 regression test cases**.

4.3.1 Industrial Adoption Rate. The most rigorous measure of generated code quality is whether engineers are willing to merge it into the master codebase[2, 23]. NL2Test achieved an impressive **Overall Adoption Rate of 85.4% (2730/3196)**, indicating that the vast majority of generated scripts required little to no modification to meet deployment standards. This high acceptance rate confirms that the generated code is not only functionally correct but also adheres to the maintainability standards required by enterprise CI/CD pipelines.

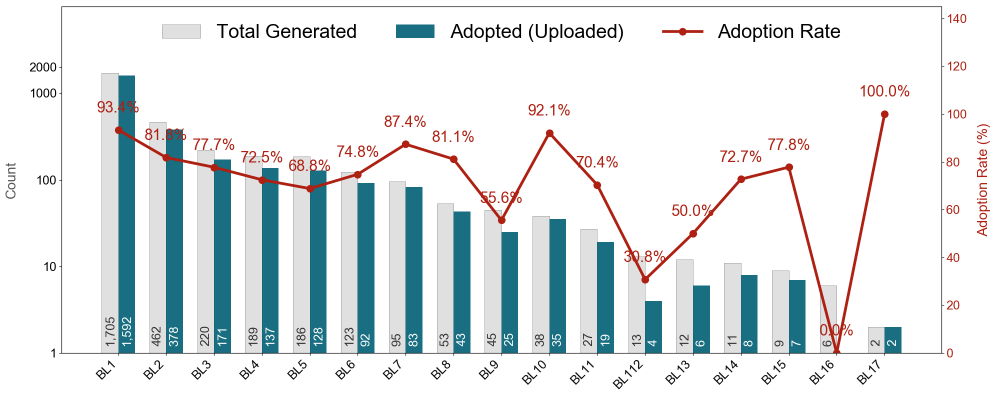


Fig. 4. Adoption Rate of Generated Cases by Business Line.

Analysis of the business lines with lower adoption rates reveals specific environmental constraints that currently limit maintainability. The *BL9* and *BL13* business lines, for instance, recorded lower

adoption rates of 55.6% and 50.0% respectively. This is primarily because these domains rely on highly dynamic, real-time content; while the tool correctly generates the request sequence, it struggles to generate precise, deterministic assertions for the rapidly changing data, forcing engineers to manually rewrite the assertion logic. Similarly, the *BL12* and *BL16* lines showed low adoption (30.8% and 0.0% respectively). These domains rely heavily on non-standard binary protocols or asynchronous callbacks rather than standard HTTP interfaces. In such cases, the denoising module fails to isolate valid business steps from the protocol overhead, resulting in low-quality scripts that engineers opted to reject rather than repair.

4.3.2 User and Expert Feedback. Complementing the quantitative adoption data, we conducted direct interviews with the leads of QA teams across 8 distinct business lines to gather their teams' overall feedback regarding the tool. Of these, **6 business lines provided explicitly positive evaluations**, emphasizing the tool's reliability and the high quality of the generated code.

Table 2. Summary of Expert Feedback and Key Metrics from Production Deployment.

Business Line	Specific Feedback
BL1	"It used to take 3 hours to write automation for a single endpoint, but now, using the LLM to generate scenarios, doing multiple endpoints takes just 1 hour. Stability is hitting the mark too—between May 30th and June 5th, the success rate for scheduled tasks was 100%."
BL2	"We saw some massive jumps in the data. P0-level API coverage is completely maxed out, going from 14.87% straight to 100%. Overall API coverage is now over halfway (at 50.68%), and the server-side recall rate increased more than fourfold (from 6.36% to 28.89%). Basically, for all features released after April, we've achieved a 'shift-left' in API automation."
BL3	"Things are going well for us. After we added cases in the first phase, the coverage saw a huge boost. It jumped straight from the previous 29.01% to 53.65%—that's nearly double."
BL4	"Both the user experience and the generation results met our expectations. A real highlight is that it supports generating complex scenarios. Data-wise, we achieved a breakthrough in automation coverage, going from 0% up to 9% (specifically, bumping the test cases from 0 to 30)."
BL5	"After using the tool, the full code coverage for P0 services shot up quickly, rising from 20% to 30% overall. At the microservice level, the biggest increase was around 40%."
BL6	"Everyone feels it's really smooth to use, and the accuracy of the generated case statements is pretty high. No major issues there."

A summary of this feedback highlights that the tool significantly enhances workflow efficiency and test stability. According to the expert reports, the generated scripts exhibit high operational stability, with the *BL1* team reporting a 100% success rate for scheduled tasks running these cases. Furthermore, the usability is described as "smooth," with high accuracy in case statements, which directly translates to reduced maintenance overhead. The feedback also notes a tangible reduction in development effort, such as the time required per scenario dropping from 3 hours to 1 hour. The specific feedback from representative business lines is detailed in Table 2. In addition, several teams reported that adopted NL2Test cases exposed real regressions after deployment, including functional logic errors, request failures or API connectivity issues, missing fields, and field-type mismatches.

4.4 RQ3: Ease of Use and Organizational Adoption

In this research question, we evaluate the tool's ease of use and its seamless integration into the daily workflow of the engineering team. We track the user journey and the volume of automation contribution over a 9-month deployment window to assess its vitality in a living environment.

4.4.1 Continuous MAU Growth. Unlike many internal tools that hit mid-stage bottlenecks[1, 20], NL2Test maintains an upward activity trend with strong penetration. Figure 5 shows that active

user growth significantly outpaced natural QA team expansion. Consequently, the monthly active user ratio rose steadily from **8.5** in April to **44.9** in December, indicating that nearly half of frontline engineers have integrated the tool into their daily workflows. This voluntary adoption, achieved without top-down mandates, suggests NL2Test drives organic growth by lowering technical barriers and addressing real engineering pain points.

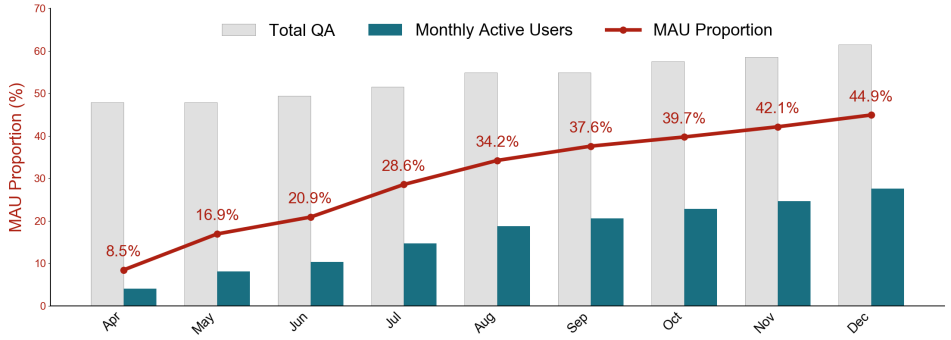


Fig. 5. Monthly Active Users Proportion Trend.

4.4.2 Proportion of LLM-Generated Test Cases. The tool successfully shifted from being an auxiliary utility to the dominant method for test creation, eventually accounting for **57% of all new test cases** generated organization-wide during its peak usage period. The timeline of this adoption reveals a distinct "Trust Latency" phenomenon[5, 27]. In the initial six months, the proportion of generated cases remained modest as engineers tested the tool on non-critical features. However, following this validation period, the share of AI-generated cases surged, crossing the majority threshold in the fourth quarter. This shift suggests a fundamental change in the organizational workflow, where engineers have moved from manually authoring tests to reviewing and approving AI-generated scripts, validating the tool's ease of use and high trust level. Internal enterprise measurements show that, as of December 2025, NL2Test saves about **30 person-days every two weeks** on average compared with manual test authoring. This result demonstrates that the tool is not only technically effective, but also practically valuable at organizational scale.

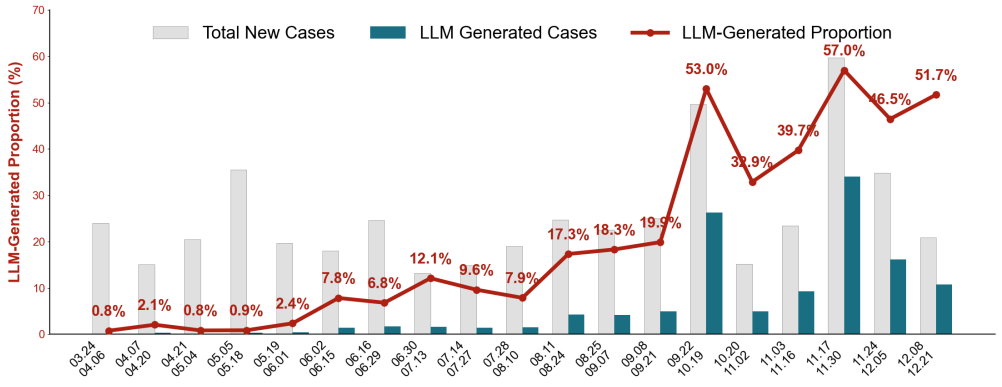


Fig. 6. Case Penetration Trend (March-Dec).

5 Lessons Learned

The development of NL2Test showed that reliable agent-based test generation depends less on larger prompts and more on limiting exact work delegated to LLM-powered agents. Many failures came from invalid intermediate artifacts, including non-existent fields, unstable assertions, incorrect bindings, and unexecutable code. We therefore adopted a constrained agent-based workflow: agents make bounded semantic decisions, while deterministic procedures validate, repair, or reject their outputs. These observations led to the following lessons.

(1) **Keep the Generation Pipeline Non-Blocking.** Early versions of NL2Test optimized each LLM stage independently, but we found that users preferred a lower-quality runnable draft to a higher-accuracy pipeline that stopped midway and required manual repair. Small structural errors could interrupt generation and make the system feel less like automation. We therefore made uninterrupted generation a design priority: uncertain LLM decisions are handled with schema checks, conservative repairs, or removal, so later stages can continue from partial but valid inputs.

(2) **Do not use the LLM for decisions that can be computed deterministically.** The LLM is useful for semantic interpretation, such as mapping business steps to requests or inferring expected outcomes, but unreliable for exact operations. In NL2Test, tasks such as field lookup, schema checking, value matching, request filtering, parameter substitution, and assertion validation are handled deterministically. LLM is used only when ambiguity or semantic reasoning is required.

(3) **Make each LLM task narrow enough to validate.** Large end-to-end generation tasks made errors difficult to locate and allowed early mistakes to propagate. We instead decomposed test generation into narrow tasks, such as request matching, dependency extraction, assertion intent extraction, and constrained code generation. Each model call produces a small structured artifact that can be validated before the next stage.

(4) **Give the model selected evidence, not the whole trace.** More context often introduced more noise. Raw traces contain static resources, polling requests, duplicated traffic, timestamps, trace identifiers, and diagnostic fields that can mislead the model. NL2Test therefore filters traces, summarizes response structures, highlights candidate fields, and excludes unstable runtime-only fields before invoking the LLM.

(5) **Constrain model outputs with allowed sets, not only natural-language instructions.** Natural-language instructions alone cannot prevent the model from inventing fields, variables, or helper functions. NL2Test provides explicit schemas and allowed sets, including valid request identifiers, candidate fields, assertion types, and existing variable names. Outputs outside these constraints are repaired only when a unique deterministic repair exists; otherwise, they are discarded.

(6) **Prefer stable partial assertions over rich but brittle assertions.** More assertions do not necessarily improve regression tests. Assertions over full responses, list order, timestamps, counters, or diagnostic messages often fail because of normal runtime variation. NL2Test therefore asserts only stable business-relevant fields and removes assertions whose targets are missing, ambiguous, or runtime-dependent.

6 Related Work

Test carving has been studied as a way to derive focused tests from higher-level executions [12]. Early work on carving and replaying differential unit tests shows that system executions can be transformed into more efficient unit-level regression tests. Recent industrial and mobile settings [6, 11, 18, 19, 30] further validate this idea, e.g., ARTISAN carves Android GUI executions into JVM

unit tests [16], and Meta's TestGen generates regression tests from observed runtime values [3]. These works establish execution-grounded test generation, but they primarily operate at the program/object level rather than at the API-traffic level.

The closest work to ours is MicroTestCarver [9], which carves E2E tests into understandable JUnit unit tests with meaningful scenarios and real test data. Our work shares its insight that E2E executions contain valuable business intent. However, MicroTestCarver targets method-level unit tests and relies on Java instrumentation, object reconstruction, and template-based generation. In contrast, our work targets API regression tests for microservice systems, using noisy HTTP traffic and a natural-language scenario, with key challenges in request selection, cross-request dependency recovery, and business assertion generation.

APICARV carves API tests from UI executions and infers OpenAPI specifications by recording and filtering browser-server traffic [33]. Unlike its specification- and coverage-oriented goal, our work focuses on scenario-based regression testing. Rather than infer general specifications, we reconstruct an executable business flow from noisy traffic, parameterize dynamic values, and generate assertions aligned with natural-language expectations.

LLM-based test generation and oracle generation are complementary to our work [13, 25, 32]. Systems such as ChatUniTest use LLMs to generate unit tests with validation and repair [7], while TOGA studies neural test oracle generation under constrained oracle spaces [10]. Our work uses LLMs differently: rather than asking an LLM to directly generate a test or oracle, we ground generation in recorded traffic and constrain it with deterministic checks for request filtering, dependency validation, and assertion-path validation. Thus, our contribution is best positioned as traffic-grounded, LLM-assisted API test carving with business-oriented assertion synthesis, rather than general LLM-based unit test generation or pure API test carving.

7 Threats to Validity

The current results should be interpreted as evidence from one industrial deployment, and the external validity of NL2Test remains uncertain. Although the benchmark spans business lines and regression scenarios, all evidence still comes from a single large-scale enterprise environment.

NL2Test is also best suited to traffic-observable API regression workflows. It assumes that a scenario can be described in natural language and that a corresponding API traffic capture is available. This assumption is natural in our deployment setting, but its applicability to environments with incomplete captures or substantially different interaction patterns remains to be studied.

8 Conclusion

This paper presents NL2Test, an agent-based workflow for generating API regression tests from natural-language scenarios and recorded traffic. Combining LLM-powered reasoning with deterministic validation, NL2Test extracts replayable workflows, reconstructs dynamic dependencies, and generates stable business-oriented assertions. Industrial evaluation and deployment show that NL2Test reduces manual effort and produces maintainable regression tests. Future work will explore stronger oracles for dynamic states and deeper CI/CD integration.

9 Data Availability

The data is not publicly available due to the inclusion of proprietary service endpoints and potentially sensitive enterprise data.

Acknowledgements

We thank the anonymous reviewers for insightful comments and suggestions. This work was supported by the National Key R&D Program of China under Grant No. 2024YFB4505902.

A Benchmark Scenarios

Table 3. Benchmark Scenarios: Natural Language Test Descriptions

ID	Natural Language (NL) Test Description
C1	**[title] Discover favorite hashtag** 1. Enter Discover page; get hashtag list. 2. Select unfavorited hashtag (collect_stat=0); click favorite [Expect] Success. 3. Check Profile hashtag list [Expect] New hashtag present.
C2	**[title] View private videos on profile page** 1. Enter Profile page. 2. Click Private tab [Expect] Private video list loads. 3. Click first video [Expect] Plays successfully.
C3	**[title] Unfavorite video** 1. Enter Profile page. 2. Click Favorites tab [Expect] Favorites list loads. 3. Select first video; click Favorite button [Expect] Unfavorited.
C4	**[title] Favorite video to a new collection** 1. Enter Feed Page. 2. Click Favorite button [Expect] Toast popup. 3. Click 'Manage' [Expect] Create Collection page. 4. Input valid name; click Save [Expect] Success.
C5	**[title] Create new collection** 1. Enter Collections page (list). 2. Click Create Collection [Expect] Creation page. 3. Input valid name; get video list. 4. Select first video; click Add Videos [Expect] Success.
C6	**[title] Delete collection** 1. Enter Collections list. 2. Select collection [Expect] Details page. 3. Click Delete Collection [Expect] Success.
C7	**[title] Cancel hashtags favorite** 1. Enter Favorited Hashtags list. 2. Select hashtag [Expect] Details page. 3. Click 'Added to Favorites' [Expect] Unfavorited. 4. Return to list [Expect] Hashtag removed.
C8	**[title] View posts list video** 1. Enter Posts list. 2. Click first video [Expect] Plays normally.
C9	**[title] Feed personalized recommendation content** 1. Open adjustment panel. 2. Select instruction; click Continue [Expect] Confirmation page. 3. Click 'Let's Go' [Expect] Success; feed refreshes.
C10	**[title] Feed delete historical personalized recommendation instruction** 1. Find video with 'Preference you set' tag. 2. Get instruction list [Expect] History page. 3. Delete first instruction [Expect] Success; feed refreshes.
C11	**[title] Feed delete historical personalized recommendation instruction** 1. Long press video; enter personalization page; get list. 2. Delete first instruction [Expect] History page updates.
C12	**[title] Feed personalized recommendation instruction** 1. Get feed video list. 2. Click 'Preference you set' tag on tagged video [Expect] History page. 3. Click '+Set New' [Expect] Adjustment panel. 4. Select instruction [Expect] Success; feed refreshes.
C13	**[title] Cancel hashtag favorite** 1. Enter Favorited Hashtag list. 2. Click hashtag [Expect] Details page. 3. Click Unfavorite [Expect] Success.
C14	**[title] Update pronoun** 1. Enter Profile. 2. Click Edit Profile. 3. Update Pronouns [Expect] status=200.
C15	**[title] Change account to private account** 1. Profile > Settings. 2. Enter Privacy. 3. Set to Private Account.
C16	**[title] Change account to BA account** 1. Profile > Settings. 2. Enter Account. 3. Switch to Business Account.
C17	**[title] Change display order of account advanced features** 1. Enter Profile. 2. Click Edit Profile. 3. Reorder features in Display Order.
C18	**[title] Account link** 1. Enter Profile. 2. Click Edit Profile. 3. Link other platform in 'Links' [Expect] status=200.
U1	**[title] Like video (logged-in user)** 1. Enter For You; get video list. 2. Like unliked video (digged=false) [Expect] Success. 3. Check Profile Likes list [Expect] Video present.
U2	**[title] Login pop-up (logged-out user)** 1. Enter Feed Page; get list. 2. Like unliked video [Expect] Login popup (3+ channels).
U3	**[title] Follow another user** 1. Enter Feed Page. 2. Follow unfollowed creator (follow_status=0) [Expect] Success (status=1). 3. Enter Following page [Expect] Creator's videos visible.
U4	**[title] Favorite video** 1. Enter Feed Page; get list. 2. Favorite unfavorited video (collect_stat=0) [Expect] Success. 3. Check Profile Favorites [Expect] List count >=1; video present.
I1	**[title] Select video template** 1. Click '+' (Shooting page). 2. Click Template (bottom right).
I2	**[title] View video effect** 1. Click video effect anchor [Expect] Effect details.
M1	**[title] Artist user selects single release, uploads & deletes full song file** 1. Click Upload > Select 'Single Song'. 2. Enter track info page. 3. Add File (upload full song). 4. Delete song file [Expect] status=200.
M2	**[title] Single upload, click submit successfully then view all releases** 1. Fill track info; click Submit. 2. Click All Releases [Expect] Song list > 0.
M3	**[title] Secondary editing and saving of releases draft** 1. Releases > Drafts. 2. Edit one of drafts; Save.
M4	**[title] After saving the draft, delete the draft on the 'My Releases' page** 1. Click Upload > Single Song. 2. Fill partial info; click 'Save to My Drafts'. 3. Releases > Drafts. 4. Delete first draft.

ID	Natural Language (NL) Test Description
M5	**[title] Contract management download and view AOP contract** 1. Click Avatar. 2. Select Contract Management. 3. Download first contract (status=200).
M6	**[title] View artist insights** 1. Releases > Artists. 2. View first artist details.
M7	**[title] Modify the title of an artist user's song 'Under Review'** 1. View All Releases. 2. View first song details. 3. Modify Title.
M8	**[title] View artist's data for the last 7 days** 1. Enter Artist Overview. 2. Select '7d' filter in Trends [Expect] Data shows 7d.
M9	**[title] View artist song list sorted by posts quantity** 1. Enter Song Performance. 2. Click 'Posts' header [Expect] Descending sort. 3. Click 'Posts' header [Expect] Ascending sort.
M10	**[title] On the artist list page, search by artist name and star an artist** 1. Enter Artist List. 2. Search 'Taylor'. 3. Click Subscribe on first result [Expect] Star=true.
M11	**[title] Artist user logs in and views songs** 1. Log in. 2. View uploaded songs [Expect] list > 0. 3. View album songs [Expect] rsp not none.
M12	**[title] Artist user modifies a song with 'Under Review' status** 1. View All Releases. 2. View unreviewed songs. 3. Edit unreviewed song; replace file. 4. Submit [Expect] Success (status=200).
M13	**[title] Artist user deletes a rejected song** 1. View All Releases. 2. Delete rejected song [Expect] Success (status=200).
M14	**[title] Label user saves a song to Drafts when uploading and then deletes it** 1. Click Upload > Enter title/lang. 2. Save to My Drafts [Expect] Success. 3. Delete draft.
M15	**[title] Label user uploads an avatar for their artists** 1. Releases > Artists. 2. Edit > Upload Avatar. 3. Submit [Expect] Success.
M16	**[title] Label user adds a new artist** 1. Releases > Artists. 2. Click Add New Artist. 3. Enter Name/Avatar. 4. Submit 5. View Artists [Expect] New Artist.
M17	**[title] User enters the song modification page without saving and returns to the homepage** 1. Releases > Edit song. 2. No changes made. 3. Click Return.
M18	**[title] Artist user logs in and changes avatar** 1. Change Avatar [Expect] Success.
M19	**[title] Creator joins WWA** 1. Studio > Work with Artists. 2. Apply [Expect] Apply Successfully.
M20	**[title] Creator submits video for WWA campaign** 1. Work with Artists > Select Campaign. 2. Click Use This Track. 3. Submit video [Expect] Success (status=200).
M21	**[title] Creator submits self-serve video** 1. Studio > Work with Artists. 2. Pass verify; Click Apply [Expect] Eligible.
M22	**[title] Creator enters WWA to view joined campaigns** 1. Studio > Work with Artists. 2. Tab: Personal Center. 3. Click Participated Campaigns [Expect] Review page.
M23	**[title] Creator enters WWA to favorite a campaign** 1. 1. Studio > Work with Artists > Campaign page. 2. Favorite campaign. 3. Check Personal Center [Expect] Success.
M24	**[title] Creator enters WWA to view bills** 1. Studio > Work with Artists. 2. Personal Center > Bills.
M25	**[title] Creator enters WWA to subscribe to a campaign** 1. Studio > Work with Artists. 2. Find Upcoming; Subscribe first one [Expect] Success.
S1	**[title] Unit conversion** 1. Search input: 'USD to CNY'.
S2	**[title] Game card** 1. Search input: 'Dota2'.

References

[1] Nadia Alshahwan, Arianna Blasi, Kinga Bojarczuk, Andrea Ciancone, Natalija Gucevska, Mark Harman, Michal Krolikowski, Rubmary Rojas, Dragos Martac, Simon Schellaert, Kate Ustiuzhanina, Inna Harper, Yue Jia, and Will Lewis. 2024. Enhancing Testing at Meta with Rich-State Simulated Populations. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice* (Lisbon, Portugal) *(ICSE-SEIP '24)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3639477.3639729

[2] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 185–196. doi:10.1145/3663529.3663839

[3] Nadia Alshahwan, Mark Harman, Alexandru Marginean, Rotem Tal, and Eddy Wang. 2024. Observation-Based Unit Test Generation at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 173–184. doi:10.1145/3663529.3663838

[4] Marcin Andrzejewski, Nina Dubicka, Jędrzej Podolak, Marek Kowal, and Jakub Siłka. 2025. Automated Test Generation Using Large Language Models. *Data* 10, 10 (2025), 156. doi:10.3390/data10100156

[5] Moritz Beller, Hongyu Li, Vivek Nair, Vijayaraghavan Murali, Imad Ahmad, Jürgen Cito, Drew Carlson, Ari Aye, and Wes Dyer. 2023. Learning to Learn to Predict Performance Regressions in Production at Meta. In *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*. 56–67. doi:10.1109/AST58925.2023.00010

[6] Xiaobao Cai, Zhen Dong, Yongjiang Wang, Abhishek Tiwari, and Xin Peng. 2024. Reproducing Timing-Dependent GUI Flaky Tests in Android Apps via a Single Event Delay. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1504–1515. doi:10.1145/3650212.3680377

[7] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 572–576. doi:10.1145/3663529.3663801

[8] Yiru Chen, Chenxi Zhang, Zhen Dong, Dingyu Yang, Xin Peng, Jiayu Ou, Hong Yang, Zheshun Wu, Xiaojun Qu, and Wei Li. 2023. Dynamic Graph Neural Networks-Based Alert Link Prediction for Online Service Systems. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 79–90. doi:10.1109/ASE56229.2023.00177

[9] Amirhossein Deljouyi and Andy Zaidman. 2023. Generating Understandable Unit Tests through End-to-End Test Scenario Carving. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 107–118. doi:10.1109/SCAM59687.2023.00021

[10] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: a neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2130–2141. doi:10.1145/3510003.3510141

[11] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 481–492. doi:10.1145/3377811.3380402

[12] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Matthew Jorde. 2009. Carving and Replaying Differential Unit Test Cases from System Test Cases. *IEEE Transactions on Software Engineering* 35, 1 (2009), 29–45. doi:10.1109/TSE.2008.103

[13] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation. *IEEE Transactions on Software Engineering* 50, 9 (2024), 2254–2268. doi:10.1109/TSE.2024.3428972

[14] Erhu Feng, Wenbo Zhou, Zibin Liu, Le Chen, Yunpeng Dong, Cheng Zhang, Yisheng Zhao, Dong Du, Zhichao Hua, Yubin Xia, et al. 2025. Get Experience from Practice: LLM Agents with Record & Replay. *arXiv preprint arXiv:2505.17716* (2025). doi:10.48550/arXiv.2505.17716

[15] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. doi:10.1145/2025113.2025179

[16] Alessio Gambi, Hemant Gouni, Daniel Berreiter, Vsevolod Tymofyeyev, and Mattia Fazzini. 2023. Action-Based Test Carving for Android Apps. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 107–116. doi:10.1109/ICSTW58534.2023.00032

[17] Jiale Guo, Suizhi Huang, Mei Li, Dong Huang, Xingsheng Chen, Regina Zhang, Zhijiang Guo, Han Yu, Siu-Ming Yiu, Pietro Lio, et al. 2025. A Comprehensive Survey on Benchmarks and Solutions in Software Engineering of LLM-Empowered Agentic System. *arXiv preprint arXiv:2510.09721* (2025). doi:10.48550/arXiv.2510.09721

[18] Wunan Guo, Zhen Dong, Liwei Shen, Wei Tian, Ting Su, and Xin Peng. 2022. Detecting and fixing data loss issues in Android apps. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 605–616. doi:10.1145/3533767.3534402

[19] Wunan Guo, Zhen Dong, Liwei Shen, Daihong Zhou, Bin Hu, Chen Zhang, and Hai Xue. 2025. Effectively Modeling UI Transition Graphs for Android Apps Via Reinforcement Learning. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. 13–24. doi:10.1109/ICPC66645.2025.00011

[20] Mark Harman, Jillian Ritchey, Inna Harper, Shubho Sengupta, Ke Mao, Abhishek Gulati, Christopher Foster, and Hervé Robert. 2025. Mutation-Guided LLM-based Test Generation at Meta. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering* (Clarion Hotel Trondheim, Trondheim, Norway) *(FSE Companion '25)*. Association for Computing Machinery, New York, NY, USA, 180–191. doi:10.1145/3696630.3728544

[21] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479* (2024). doi:10.48550/arXiv.2408.02479

[22] Myeongsoo Kim, Tyler Stennett, Dhruv Shah, Saurabh Sinha, and Alessandro Orso. 2024. Leveraging Large Language Models to Improve REST API Testing. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (Lisbon, Portugal) *(ICSE-NIER'24)*. Association for Computing Machinery, New York, NY, USA, 37–41. doi:10.1145/3639476.3639769

[23] Yinfeng Li, Chen Gao, Xiaoyi Du, Huazhou Wei, Hengliang Luo, Depeng Jin, and Yong Li. 2022. Automatically Discovering User Consumption Intents in Meituan. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 3259–3269. doi:10.1145/3534678.3539122

[24] Cristian Mascia, Antonio Guerriero, Luca Giamattei, Roberto Pietrantuono, and Stefano Russo. 2025. Microservices Performance Testing with Causality-enhanced Large Language Models. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. 136–140. doi:10.1109/Forge66646.2025.00022

[25] Facundo Molina, Alessandra Gorla, and Marcelo d'Amorim. 2025. Test Oracle Automation in the Era of LLMs. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 150 (May 2025), 24 pages. doi:10.1145/3715107

[26] Dezhi Ran, Hao Wang, Zihe Song, Mengzhou Wu, Yuan Cao, Ying Zhang, Wei Yang, and Tao Xie. 2024. Guardian: A Runtime Framework for LLM-Based UI Exploration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 958–970. doi:10.1145/3650212.3680334

[27] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. 2023. ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 969–985. https://www.usenix.org/conference/osdi23/presentation/saokar

[28] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. 2024. TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 30–34. doi:10.1145/3639478.3640024

[29] Sumedh Sathaye, Patrick East, Reut Kovetz, Jennifer Minarik, Kelly Lisai, Arthur Lent, and Nicole Reineke. 2023. Creating CLI packages and API playbooks from codified graphical user experience designs. US Patent 11,740,878.

[30] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and finding system setting-related defects in Android apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 204–215. doi:10.1145/3460319.3464806

[31] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, Nikhil Collooru, Ke Wang, Sameer Agarwal, Arjun Gupta, Dionysios Logothetis, Kostas Xirogiannopoulos, Amit Dutta, Varun Gajjala, Rohit Jain, Ajay Palakuzhy, Prithvi Pandian, Sergey Pershin, Abhisek Saikia, Pranjal Shankhdhar, Neerad Somanchi, Swapnil Tailor, Jialiang Tan, Sreeni Viswanadha, Zac Wen, Biswapesh Chattopadhyay, Bin Fan, Deepak Majeti, and Aditi Pandit. 2023. Presto: A Decade of SQL Analytics at Meta. *Proc. ACM Manag. Data* 1, 2, Article 189 (June 2023), 25 pages. doi:10.1145/3589769

[32] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936. doi:10.1109/TSE.2024.3368208

[33] Rahulkrishna Yandrapally, Saurabh Sinha, Rachel Tzoref-Brill, and Ali Mesbah. 2023. Carving UI Tests to Generate API Tests and API Specification. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1971–1982. doi:10.1109/ICSE48619.2023.00167

[34] Chenxi Zhang, Zhen Dong, Xin Peng, Bicheng Zhang, and Miao Chen. 2024. Trace-based Multi-Dimensional Root Cause Localization of Performance Issues in Microservice Systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 110, 12 pages. doi:10.1145/3597503.3639088

[35] Quanjun Zhang, Weifeng Sun, Chunrong Fang, Bowen Yu, Hongyan Li, Meng Yan, Jianyi Zhou, and Zhenyu Chen. 2025. Exploring Automated Assertion Generation via Large Language Models. *ACM Trans. Softw. Eng. Methodol.* 34, 3, Article 81 (Feb. 2025), 25 pages. doi:10.1145/3699598

Received 2026-01-30; accepted 2026-06-25