JSTestCraft: Addressing Context Deficits in JavaScript Unit Test Generation via Agentic Multi-Level Contextual Analysis

YIYANG LIU, Huazhong University of Science and Technology, China
YANJIE ZHAO*, Huazhong University of Science and Technology, China
HAOYU WANG, Huazhong University of Science and Technology, China

Unit testing is crucial for software reliability in JavaScript, yet constructing comprehensive suites remains costly and error-prone. JavaScript's dynamic features, flexible typing, asynchronous execution, and reliance on third-party libraries, pose persistent challenges for automated test generation. Although large language models (LLMs) show promise in code reasoning and synthesis, existing methods often miss the multilayer contextual information required for executable, semantically correct tests.

This paper introduces JSTestCraft, an agentic, multi-layer framework for adaptive JavaScript unit test generation. JSTestCraft reconstructs missing context via three enrichment agents: library, structural, and semantic. These agents capture inter-function topology, third-party API semantics, and inferred type constraints in a shared contextual memory for reasoning-driven test synthesis. A testing and optimization layer generates, executes, and iteratively refines test cases. Evaluated on 20 real-world Node.js repositories, JSTestCraft outperforms state-of-the-art baselines, improving test pass rate by **60.9%**, statement coverage by **14.2%**, and branch coverage by **47.3%**. Ablation analysis confirms each agent's contribution to contextual completeness. Beyond metrics, JSTestCraft discovers 13 previously unknown bugs across 5 repositories, 6 of which received substantive acknowledgment from maintainers or community contributors. These results demonstrate that context reconstruction and agentic collaboration enable LLMs to perform more reliable and adaptive testing in dynamic JavaScript environments.

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

Additional Key Words and Phrases: JavaScript, unit test generation, large language models, multi-agent systems

ACM Reference Format:
Yiyang Liu, Yanjie Zhao, and Haoyu Wang. 2018. JSTestCraft: Addressing Context Deficits in JavaScript Unit Test Generation via Agentic Multi-Level Contextual Analysis. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 34 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Unit testing has become a cornerstone of modern software development, serving as the first line of defense in ensuring software quality and reliability [7]. In agile and continuous integration environments, unit tests not only verify functional correctness but also enable early defect detection, significantly reducing maintenance costs and facilitating collaboration [15]. This has led to mature frameworks such as JUnit [31] for Java, PyTest [47] for Python, and Mocha [38]

*Corresponding author.

Authors' Contact Information: Yiyang Liu, Huazhong University of Science and Technology, Wuhan, China, yiyangliu@hust.edu.cn; Yanjie Zhao, Huazhong University of Science and Technology, Wuhan, China, yanjie_zhao@hust.edu.cn; Haoyu Wang, Huazhong University of Science and Technology, Wuhan, China, haoyuwang@hust.edu.cn.

for JavaScript. However, writing effective unit tests remains both time-consuming and cognitively demanding [60], especially for dynamically typed and event-driven languages such as JavaScript.

JavaScript poses distinctive and persistent challenges for unit test generation. Its dynamic typing allows runtime type mutation, making static reasoning about program behavior and data flow difficult [45]. Asynchronous patterns such as callbacks, promises, and `async/await` introduce highly non-linear execution flows that complicate dependency tracing and state consistency [50]. Moreover, its vast and rapidly evolving ecosystem of third-party libraries introduces additional uncertainty: API semantics may evolve, documentation may lag, and dependencies are often intertwined in subtle and implicit ways [61]. These properties make JavaScript testing a challenging and distinctive frontier where both traditional and modern methods struggle to generate executable and semantically valid tests.

Existing methods fall short due to a fundamental limitation: the lack of contextual understanding. Traditional approaches, such as Randoop [44] and KLEE [2], rely primarily on path exploration or mutation to achieve coverage. While effective for structural testing, they fail to capture the higher-level semantics governing how functions interact or how APIs behave [57]. Recently, large language models (LLMs) have been introduced for test generation [16, 30, 54], demonstrating remarkable progress in producing human-readable and semantically meaningful tests. Yet, despite their linguistic fluency, current LLM-based systems still suffer from what we term the **context deficit problem**: they generate tests without sufficient grounding in the broader code, type, or API context. Empirical observations reveal that this deficit manifests in three major forms: (1) misunderstanding third-party APIs, (2) failing to relate correlated synchronous/asynchronous functions, and (3) constructing invalid test inputs due to incomplete type reasoning. Ultimately, the quality of generated tests depends not only on model scale or decoding strategy, but on the *richness and adaptivity of contextual information* provided to the model.

To illustrate the multifaceted nature of this context deficit, consider the following motivating example. A function that uses the `color-hash` library requires understanding of external API semantics, an instance of *API context*. If the same module defines both `writeFile` and `writeFileSync`, they form a synchronous/asynchronous pair, where effective testing requires knowledge of their structural correlation, *structural context*. Finally, functions such as `normalizePort`, which accept dynamically typed arguments, demand reasoning about runtime type constraints, *semantic context*. Missing any of these dimensions results in failed, incomplete, or semantically meaningless test cases. Therefore, the central challenge in JavaScript unit test generation is not merely exploring program space, but enriching the contextual foundation upon which test generation is performed. To maintain focus, we defer code-level illustrations of each context type to Section 2, where three corresponding case studies concretely demonstrate these phenomena.

We present JSTestCraft, a multi-agent context-enrichment framework designed to address the complex challenges of automated test generation in JavaScript. The concept of *agents* as autonomous, reactive, and proactive software components that collaborate through shared context has a long history in multi-agent systems research [55, 56], predating the recent popularization of "LLM-based agents" in contemporary AI discourse. JSTestCraft reimagines the test generation process as a coordinated pipeline of specialized *agents*, each responsible for reconstructing a specific and complementary dimension of program context. The **Library Enrichment Agent** autonomously retrieves and interprets third-party API semantics, constructing knowledge graphs to represent and reason about external dependencies. The **Structural Enrichment Agent** analyzes inter-function relationships, identifies synchronous and asynchronous variants, and detects shared behavioral patterns that guide the generation of correlated and semantically consistent test cases. The **Semantic Enrichment Agent** performs dynamic type inference and constraint reasoning to improve input synthesis for loosely typed or under-specified functions. Rather than engaging in a sequential reasoning loop, these enrichment agents operate in a *parallel and cooperative* fashion, each contributing complementary insights

to a shared contextual memory. This aggregated context is subsequently consumed by the downstream **Testing & Optimization Agent**, which orchestrates an execution-grounded reasoning loop and iteratively refines test quality based on runtime feedback and coverage analysis. Together, these components establish an agentic paradigm of perception, enrichment, and adaptive reasoning for context-aware and reliable automated test generation.

Our key contributions are summarized as follows:

- We provide the first systematic characterization of *context deficits* in JavaScript test generation, identifying three complementary dimensions of missing information: external, structural, and semantic. We further conduct empirical analyses to quantify their collective impact on test effectiveness.
- We design and implement JSTestCraft, a *multi-agent*, retrieval-augmented framework composed of three cooperative agents, namely the **Library Enrichment Agent**, **Structural Enrichment Agent**, and **Semantic Enrichment Agent**, which enable adaptive context acquisition, contextual reasoning, and test synthesis.
- We evaluate JSTestCraft on 20 diverse Node.js repositories and show that it improves test pass rate by 60.9%, statement coverage by 14.2%, and branch coverage by 47.3% compared with the state-of-the-art baseline *TestPilot*. In addition, JSTestCraft autonomously discovers 13 previously unreported bugs, 6 received substantive acknowledgment from maintainers or community contributors.

Artifact Availability. The artifacts of this paper are available at https://figshare.com/s/84dd5b95e69a9ebbecc6.

2 The Context Deficit Problem in JavaScript Testing

Despite significant advances in automated test generation, existing approaches continue to underperform on JavaScript programs due to their inability to capture the multi-layered contextual information required for producing reliable and executable tests. Through empirical analysis across diverse open-source projects, we observe that failures in generated tests often stem not from algorithmic inadequacy but from the absence of contextual grounding in how JavaScript code actually behaves. We refer to this as the *context deficit problem*, which manifests across three interdependent dimensions: *external*, *structural*, and *semantic*. Each dimension degrades the effectiveness of generated tests in complementary ways.

2.1 Overview: A Taxonomy of Context Deficits

Figure 1 illustrates the interplay among these three dimensions and how their mutual dependencies collectively shape the outcome of JavaScript test generation. The *external* context captures knowledge of third-party library APIs and their behavioral semantics. Without such understanding, generators may misuse imported modules or misinterpret API contracts. The *structural* context represents the relationships among correlated functions, such as synchronous/asynchronous variants or wrapper–core pairings, whose absence causes fragmented reasoning and inconsistent behavioral validation. Finally, the *semantic* context concerns the inference of valid runtime types and input constraints in dynamically typed environments, which determines the executability and precision of generated tests.

Crucially, these three dimensions are *not independent*. Their interdependencies arise from the fundamental ways in which JavaScript code is structured, typed, and composed, and are consistently observed across the real-world projects analyzed in this study. As illustrated in Figure 1, deficiencies propagate across dimensions in a cyclical and mutually reinforcing manner.

The first propagation path runs from *external* to *structural* context. When a generator lacks knowledge of a third-party API's behavioral contract, for instance, not knowing that `ColorHash.prototype.hex` requires prior object instantiation, it cannot determine the lifecycle dependency between `getUserColor` and the `ColorHash` constructor. This is not a

coincidental failure: without knowing that `hex()` is an instance method, the generator has no basis to infer that a `ColorHash` object must be created and maintained before the function can be invoked. The structural linkage between the two components is therefore not merely obscured but rendered invisible, as the missing API contract is the only evidence from which that dependency could have been recovered.

The second propagation path runs from *structural* to *semantic* context. Consider `writeFile` and `writeFileSync`, which share an identical `options` parameter interface. The full type constraints on this shared parameter, which fields it accepts, which are optional, and what types they expect, can only be reliably inferred by cross-referencing both functions as behavioral variants. When structural context is absent and each function is analyzed in isolation, the generator observes the `options` parameter only within the narrow scope of a single function body, where its usage is often incomplete or implicit. The type inference problem becomes severely under-constrained: a single function slice simply does not contain enough evidence to determine the full type domain of a parameter whose contract is defined jointly across multiple correlated functions.

The third propagation path runs from *semantic* back to *external* context. When `val` in `normalizePort` is incorrectly inferred as always numeric, the generator treats `normalizePort(3000)` as the only valid invocation pattern. This flawed assumption does not remain local: any downstream library call that passes the return value of `normalizePort` as an argument will be constructed with a numeric type, even when the receiving API expects a `string | number` union. The semantic error propagates outward and actively corrupts the construction of external API calls throughout the test suite, completing the cycle.

This cyclical propagation explains why remediating any single dimension in isolation remains insufficient: enriching only API knowledge, or only type information, leaves the other dimensions vulnerable to injecting errors that undermine the enriched one. Effective JavaScript test generation therefore requires holistic, multi-level context enrichment that simultaneously addresses all three perspectives.

Together, these three contextual layers define the cognitive foundation upon which automated test generation must operate. The following subsections provide a detailed analysis of each deficit, supported by representative empirical cases drawn from real-world JavaScript projects.

2.2 External Context Deficit: Third-Party Library Semantics

2.2.1 Problem Characterization. JavaScript applications are deeply intertwined with the npm ecosystem, relying extensively on third-party libraries [25]. However, automated test generators typically treat these external dependencies as opaque modules. Static analysis tools can only extract superficial function signatures, while LLM-based generators may rely on incomplete or outdated prior knowledge [13, 61]. This results in a lack of understanding about external API semantics, parameter conventions, and error-handling behaviors.

2.2.2 Motivating Example. Figure 2 illustrates this problem through the `color-hash` case, where the generator, lacking complete knowledge of the API's calling conventions, incorrectly assumes `hex()` to be a static method rather than an instance method. One might expect that the function under test itself, which invokes `colorHash.hex(name)`, could serve as an in-context example of correct usage. However, this does not hold in practice: `colorHash` is declared at the module level rather than within the function body, and our function slicing stage extracts only the function's lexical scope and its direct dependencies, leaving the module-level instantiation invisible to the generator. More broadly, this case represents only the simplest form of external context deficit; in more complex scenarios where API calls are distributed across files or functions receive already-instantiated objects as parameters, the function body provides no

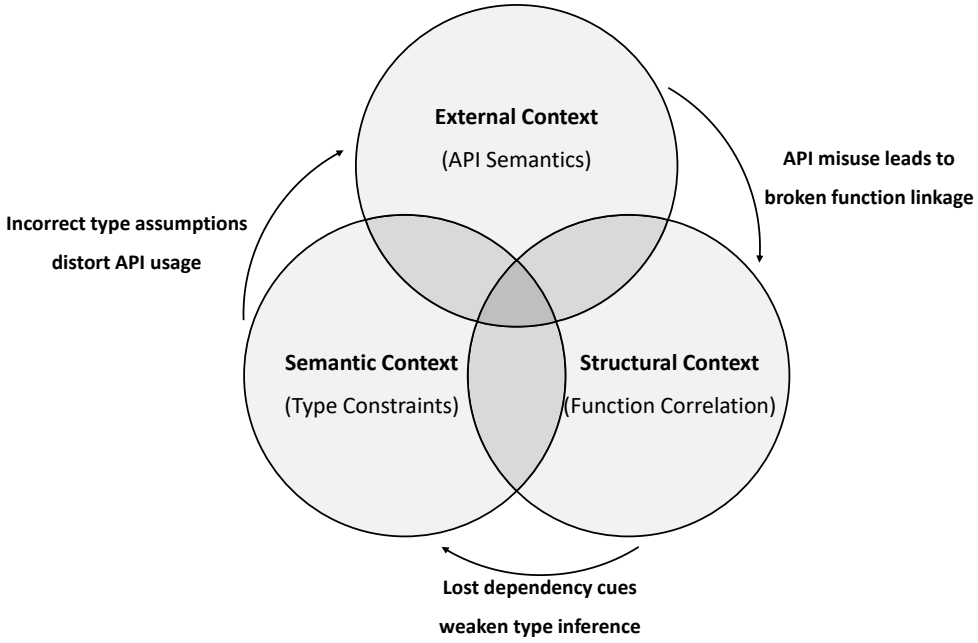

Fig. 1. Interdependence among the three dimensions of the *context deficit problem*. The arrows indicate propagation of deficiencies: (1) missing API semantics (*external*) obscure inter-function reasoning (*structural*); (2) incomplete structural understanding hinders valid type inference (*semantic*); and (3) incorrect semantic assumptions distort external API usage (*external*).

usage example at all. As a result, the generated test is syntactically valid but semantically incorrect, as it bypasses the required object instantiation step. Notably, this error persists even under iterative execution feedback: the runtime error message indicates only that the call failed, but provides no information about the need for prior instantiation. The missing knowledge cannot be recovered from execution signals alone, but only from accurate API documentation or semantic grounding. This observation directly motivates the Library Enrichment Agent, which explicitly retrieves and injects such calling-convention knowledge before test generation. Furthermore, existing approaches such as TestPilot, which rely on static heuristics and implicit LLM priors without dedicated API semantic retrieval, similarly fail to generate correct tests in this scenario, as they share the same fundamental limitation of treating third-party dependencies as opaque modules.

2.2.3 Reasons for the Failure of Existing Approaches. Coverage-driven tools treat APIs as black boxes, while LLM-based approaches rely on implicit memory of library usage, which may be inaccurate or outdated. When documentation is missing or ambiguous, these systems cannot autonomously recover semantic meaning, and execution feedback alone is insufficient to supply the missing API knowledge. This motivates the need for an agentic library reasoning module that can retrieve, interpret, and refine API semantics before test generation.

2.3 Structural Context Deficit: Inter-Function Dependencies

2.3.1 Problem Characterization. JavaScript modules frequently define multiple correlated functions that together implement a single logical behavior, yet existing test generators analyze and generate tests for each function in

```javascript
const ColorHash = require('color-hash');
const colorHash = new ColorHash();

function getUserColor(name) {
  return colorHash.hex(name);
}

module.exports = getUserColor;
```

(a) Function utilizing the `color-hash` library.

```javascript
const assert = require('assert');
const getUserColor = require('./getUserColor');
const ColorHash = require('color-hash');

// Incorrect usage: hex() is called as if it's static
const color = ColorHash.hex('alice');

assert.match(color, /^#[0-9a-fA-F]{6}$/);
```

(b) Incorrect API invocation in DeepSeek-v3 generated test due to incomplete prior knowledge of calling conventions.

Fig. 2. Case study demonstrating LLM failures in handling third-party library interfaces under zero-shot generation without API semantic grounding.

isolation, ignoring their interdependencies and shared invariants [26]. This structural context deficit manifests across three distinct scenarios. The first involves *synchronous and asynchronous variants*, where two functions implement identical logic but differ in their execution model, requiring coordinated testing to verify behavioral consistency across both implementations. The second involves *file-level co-located functions*, where multiple functions within the same module share common state, initialization logic, or error-handling invariants, such that testing any one function independently fails to capture their collective behavioral contracts. The third involves *naming-pattern clusters*, where functions sharing a common naming convention, such as `parseX`, `validateX`, and `formatX`, implement tightly coupled stages of the same processing pipeline, and isolated testing of each stage overlooks the dependencies between them. While all three scenarios represent structural context deficits, the following motivating example focuses on the synchronous/asynchronous case, as it most concisely illustrates the cost of ignoring inter-function dependencies.

2.3.2 Motivating Example. Figure 3 shows two related Node.js functions, `fs.writeFile` and `fs.writeFileSync`, which perform the same logical operation of writing serialized objects to disk but differ in their execution model. A generator that treats them independently may verify only one function or produce redundant tests that fail to check consistency across both implementations. It is worth clarifying that JSTestCraft does not replace independent testing with structural testing: each function is still tested independently to verify its individual correctness. The structural context introduces additional testing dimensions that independent testing cannot cover, varying by relationship type. For synchronous and asynchronous variants such as `writeFile` and `writeFileSync`, the additional dimension is cross-function behavioral consistency, both should produce identical output given the same input and handle error conditions in the same way. For file-level co-located functions, the additional dimension is shared state and initialization correctness. For naming-pattern clusters, the additional dimension is end-to-end correctness across tightly coupled processing stages. Figure 3(b) illustrates how JSTestCraft leverages structural context to target this additional dimension: by

recognizing `writeFile` and `writeFileSync` as behavioral variants sharing the same logical contract, JSTestCraft prompts DeepSeek-v3 to generate a coordinated test that invokes both implementations and asserts their output equivalence, capturing the consistency property that isolated testing would miss.

```
async function _writeFile (file, obj, options = {}) {
  const fs = options.fs || _fs
  const str = stringify(obj, options)

  await universalify.fromCallback(fs.writeFile)(file, str, options)

}
const writeFile = universalify.fromPromise(_writeFile)


function writeFileSync (file, obj, options = {}) {
  const fs = options.fs || _fs
  const str = stringify(obj, options)

  return fs.writeFileSync(file, str, options)
}
```

(a) Related synchronous and asynchronous file-writing functions.

```
const data = { name: "Alice", age: 30 };
const filePath = "./test.json";


await writeFile(filePath, data);
const contentA = fs.readFileSync(filePath, "utf-8");


writeFileSync(filePath, data);
const contentB = fs.readFileSync(filePath, "utf-8");


assert.strictEqual(contentA, contentB);
```

(b) Integrated test case generated by JSTestCraft via structural context enrichment.

Fig. 3. Case study demonstrating the need for coordinated testing of interrelated functions.

2.3.3 The Cost of Isolated Testing. Without structural context, generated tests cannot verify behavioral equivalence, shared edge cases, or temporal dependencies. This leads to fragmented verification where tests pass individually but fail to ensure module-level correctness. A correlation-aware module is therefore required to identify and jointly test functionally related components, enabling unified reasoning about their interactions.

2.4 Semantic Context Deficit: Dynamic Type Constraints

2.4.1 Problem Characterization. JavaScript's dynamic typing system introduces uncertainty in parameter domains and return types [29, 52]. Test generators must infer input and output types from limited context, often leading to invalid assumptions, unexercised branches, or spurious assertions.

2.4.2 Motivating Example. The `normalizePort` case in Figure 4 demonstrates the semantic deficit: without inferring that the input may be either a string or number, the generator constructs incomplete test inputs and fails to trigger key branches. This highlights the difficulty of reasoning about runtime types in dynamically typed languages.

```
function normalizePort(val) {
  const port = parseInt(val, 10);
  if (isNaN(port)) return val;
  if (port >= 0) return port;
  return false;
}

module.exports = normalizePort;
```

(a) Function demonstrating dynamic input type handling.

```
const assert = require('assert');
const normalizePort = require('./normalizePort');

assert.strictEqual(normalizePort(3000), 3000);

// Incorrect usage: assumes input is always number
assert.strictEqual(
  normalizePort('not-a-number’),
  false);
```

(b) Type inference errors in DeepSeek-v3 generated test.

Fig. 4. Case study illustrating challenges in dynamic type inference for JavaScript functions.

2.4.3 Inference Challenges in Dynamic Languages. Type inference in JavaScript requires reasoning beyond static syntax, considering function usage patterns, runtime coercion rules, and contextual dependencies. To address these issues, an agentic type inference module can integrate lightweight static analysis with LLM-guided probabilistic reasoning, improving both coverage and test validity.

2.5 Summary: The Need for Multi-Level Context Enrichment

The three case studies presented in Sections 2.2, 2.3, and 2.4 collectively demonstrate that context deficits in JavaScript test generation are not isolated phenomena but form a mutually reinforcing cycle. As established in Section 2.1, this cycle operates through three empirically grounded propagation paths: missing API knowledge (*external*) renders inter-function dependencies invisible (*structural*), as demonstrated by the `color-hash` case where the absence of instance-method semantics severs the structural linkage between `getUserColor` and its `ColorHash` dependency; incomplete structural understanding under-constrains type inference (*semantic*), as demonstrated by the `writeFile`/`writeFileSync` case where unrecognized behavioral variants leave the shared `options` parameter type domain unresolvable from any single function slice; and incorrect semantic assumptions corrupt external API call construction (*external*), as demonstrated by the `normalizePort` case where a flawed numeric type inference propagates outward into downstream library invocations throughout the test suite.

These observations motivate the design of JSTestCraft, a modular and agentic framework that performs **multi-agent context enrichment** through parallel, cooperative agents working on distinct contextual dimensions. By simultaneously reconstructing the external, structural, and semantic contexts, JSTestCraft establishes a comprehensive foundation that enables large language models to perform grounded, executable, and semantically meaningful unit test generation for real-world JavaScript projects.

3 Approach

To overcome the context deficit problem described in Section 2, we design JSTestCraft, an agentic, multi-layer framework that systematically reconstructs missing contextual information and leverages it for reliable, executable, and iteratively optimized JavaScript unit test generation. As shown in Figure 5, JSTestCraft operates as a closed feedback system composed of two cooperative layers: (1) a **Context Enrichment Layer**, responsible for recovering multi-dimensional context, and (2) a **Testing & Optimization Layer**, which synthesizes, executes, and refines unit tests using the reconstructed contextual memory.

3.1 System Overview

JSTestCraft organizes the testing workflow into an agentic pipeline, with each layer encompassing specialized reasoning agents that perform complementary tasks. The entire process unfolds in three macro phases: 1. Function identification and slicing, 2. Multi-agent context enrichment, and 3. Adaptive test generation and optimization.

After completing the function identification and slicing phase, where individual functions and their boundaries are identified, JSTestCraft transitions into the multi-agent context enrichment phase. At this stage, the system moves from isolated function analysis to contextual reconstruction, with multiple agents collaborating to recover the missing external, structural, and semantic context. This enriched contextual knowledge forms the foundation for the final test generation and optimization phase.

The Context Enrichment Layer is composed of three specialized agents:

(1) The **Library Enrichment Agent (LEA)**, which captures and summarizes the semantics of third-party APIs used in the project.
(2) The **Structural Enrichment Agent (SEA)**, which rebuilds inter-function relationships, identifies correlated function variants, and abstracts dependency clusters.
(3) The **Semantic Enrichment Agent (SmEA)**, which performs type inference and constraint reasoning to enrich runtime semantics.

These three agents work together to populate a Shared Context Memory (SCM), which consolidates their findings into a unified knowledge base that serves as input for the downstream reasoning in the Testing & Optimization Layer.

The Testing & Optimization Layer, driven by the **Testing & Optimization Agent (TOA)**, consumes the enriched SCM, composes structured prompts for LLM-based test synthesis, and refines the generated tests through iterative feedback, improving coverage and execution validity.

3.2 Function Identification and Slicing

This preliminary stage prepares the raw material for multi-agent reasoning by systematically identifying testable functions and extracting their minimal executable slices. Rather than introducing novel static analysis algorithms, we leverage established program analysis frameworks such as Babel and Joern to perform precise syntax tree traversal and

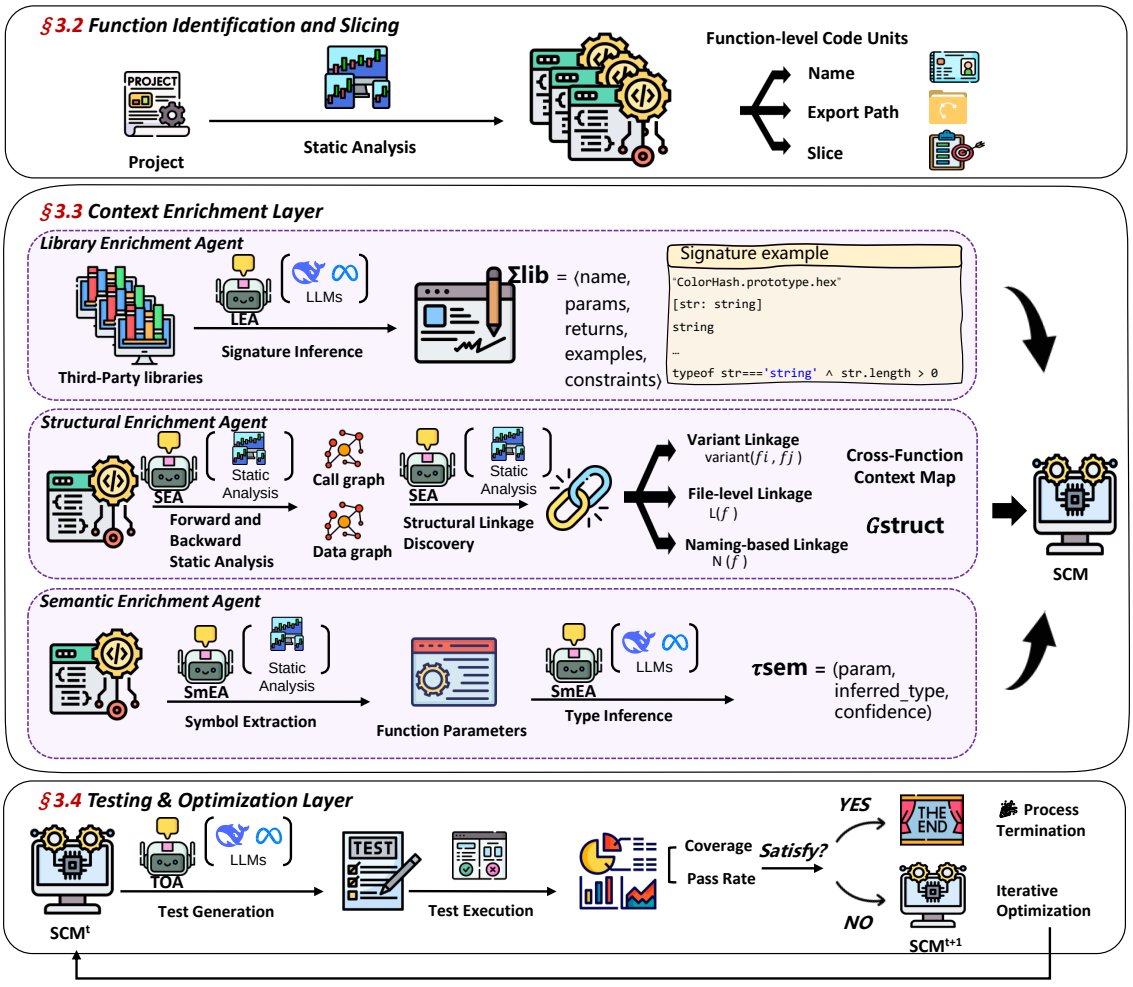

Fig. 5. Overview of JSTestCraft. The Context Enrichment Layer comprises three collaborating agents (LEA, SEA, and SmEA) that continuously populate a Shared Context Memory (SCM). The Testing & Optimization Layer, driven by a dedicated agent, consumes SCM to generate, execute, and refine tests, feeding runtime feedback back into SCM for adaptive improvement.

scope resolution. The objective here is not algorithmic innovation in parsing but to ensure accurate, language-aware extraction that faithfully preserves control and data dependencies essential for subsequent contextual reconstruction.

For each JavaScript file, we extract functions, class methods, and object methods across major definition patterns, including named, anonymous, arrow, and class forms. All direct and indirect export bindings (e.g., via `export default`, `module.exports`) are normalized to ensure external visibility. Each function f is represented as:

$$f = \langle name, slice(f), export\_path \rangle \tag{1}$$

where `slice(f)` includes its lexical scope, relevant variable declarations, and dependent imports. The heuristics applied in function export resolution are summarized in Table 1, which outlines strategies for resolving various export cases

encountered during the function identification and slicing phase. These heuristics address common challenges such as handling duplicate function definitions, aliased exports, and dynamic assignments, ensuring that all relevant functions are correctly extracted and associated with their respective export paths.

Table 1. Heuristics applied in function export resolution.

Case	Handling Strategy
Duplicate function definitions	Apply disambiguation based on structural completeness and export priority to preserve canonical definitions.
Aliased exports	Track variable references across lexical scopes to resolve indirect bindings (e.g., `const helper = utils.helper; exports.help = helper`).
Object-level exports	Recursively inspect object properties to extract callable members while filtering non-functions (e.g., `module.exports = {fn1, fn2, data}`).
Higher-order exports	Analyze function return types and nested definitions to identify exportable functions within closures (e.g., `exports.factory = () => innerFunc`).
Dynamic assignments	Employ control flow analysis to trace runtime property assignments and resolve dynamic export paths (e.g., `exports[key] = func`).
ES6 destructuring	Parse destructuring patterns in export statements to map elements to function definitions (e.g., `export {fn1 as alias, fn2}`).

3.3 Context Enrichment Layer

The Context Enrichment Layer forms the cognitive core of JSTestCraft, transforming static slices into a unified, knowledge-rich representation. Three specialized agents operate asynchronously: LEA, SEA, and SmEA. Each agent reconstructs a distinct form of missing context and contributes its output to the SCM. Through their continuous cooperation, library knowledge, structural insights, and semantic reasoning reinforce one another, enabling the system to recover the contextual completeness required for reliable test synthesis.

3.3.1 Library Enrichment Agent. The Library Enrichment Agent (LEA) captures the *external context*, which refers to the semantic knowledge of third-party libraries that the LLM itself may not fully possess. Modern JavaScript projects rely heavily on imported dependencies, and incorrect API usage has been repeatedly identified as one of the major sources of test generation failures. To address this issue, LEA systematically analyzes project dependencies, reconstructs library interfaces, and infers their behavioral semantics through structured reasoning and reflection-based inspection.

The complete extraction pipeline is illustrated in Algorithm 1. The process begins with dependency collection, where both the `package.json` file and all source files are parsed to detect static as well as dynamic import statements. LEA does not analyze all exported methods of all external libraries indiscriminately: it only processes libraries that are both declared in package.json and actively imported in the project's source files, with Node.js built-in modules explicitly excluded. For each detected package l_k, LEA executes a sandboxed loader that safely enumerates exported entities via reflection functions such as `Object.keys()`. It is worth noting that this reflection-based approach has inherent limitations for libraries that rely on native C++ extensions, environment-specific configurations, or non-standard module structures. LEA handles such failures gracefully: a failed library load returns an empty export map and the affected package is skipped without interrupting the pipeline, with the system falling back to the LLM's implicit prior knowledge for that dependency. Callable entities are then filtered and processed to extract function-level signatures.

To bridge the gap between syntactic information and deeper semantic understanding, LEA integrates reasoning with a large language model to infer parameter meanings, return values, representative usage examples, and potential constraints, forming a structured function signature defined as:

$$\Sigma_{\text{lib}} = \langle \text{name, params, returns, examples, constraints} \rangle. \tag{2}$$

Each extracted signature is transformed into markdown text, embedded using LangChain with FAISS, and stored in a persistent retrievable corpus Σ_{lib} that supports subsequent test synthesis. Furthermore, the resulting signature corpus is persistently stored and incrementally reusable across multiple test generation sessions: once a library's signatures have been extracted and indexed, they can be directly retrieved in all subsequent runs without re-analysis, effectively amortizing the extraction cost over time and avoiding redundant computation. During prompt construction, the system retrieves the top-k most relevant signatures corresponding to the target function, enabling the LLM to reference accurate and up-to-date third-party semantics even when they are underrepresented in its pretraining corpus. This process enhances both the correctness and diversity of generated tests, preventing invocation errors and enabling the synthesis of more realistic assertions that reflect actual library behaviors.

Algorithm 1: Third-Party Function Signature Extraction

Input: Project directory with `package.json`
Output: Library signature corpus $\Sigma_{\text{lib}} = \langle \text{name, params, returns, examples, constraints} \rangle$

Dependency Collection:
Parse `package.json` and all source files to extract dependency set L;
foreach *source file* s **do**
  Parse AST of s and collect all `import`/`require` modules;
  foreach *module* m **do**
    Normalize package name;
    if $m \in$ *dependencies and* $m \notin$ *Node.js built-ins* **then**
      Add m to L;

Interface Extraction:
foreach *library* $l_k \in L$ **do**
  Load l_k in a sandboxed environment;
  Enumerate exports $E =$ `Object.keys(require(`l_k`))`;
  foreach *callable symbol* $f \in E$ **do**
    Perform lightweight AST traversal to capture call pattern;
    Invoke LLM to infer `params`, `returns`, `examples`, and `constraints`;
    Construct structured record $\Sigma_f = \langle \text{name, params, returns, examples, constraints} \rangle$;
    Append Σ_f to corpus Σ_{lib};

Embedding and Indexing:
Convert Σ_{lib} entries into Markdown-formatted text;
Embed using LangChain + FAISS for retrieval during test synthesis;
return Σ_{lib}

3.3.2 Structural Enrichment Agent. The Structural Enrichment Agent (SEA) focuses on reconstructing the *structural context* that is often lost when functions are analyzed independently. Starting from the function slices identified in the previous stage, SEA rebuilds inter-function relationships, dependency chains, and higher-level linkages such as synchronous and asynchronous variants. Rather than depending on handcrafted rules, SEA leverages existing program

analysis frameworks such as Babel's traverse API to efficiently extract function invocations, data flows, and structural dependencies. Specifically, SEA first constructs a call graph and a data flow graph through forward and backward static analysis, capturing direct invocation relationships and inter-function data dependencies respectively. These two graphs serve as the prerequisite foundation for the subsequent structural linkage discovery: the call graph provides invocation topology that informs variant detection and file-level clustering, while the data flow graph reveals how data propagates across function boundaries, enabling more precise identification of naming-based and behavioral relationships.

SEA models the project's functional associations as a directed function graph G_{struct}, where nodes represent individual functions and edges represent functional association relationships inferred from variant linkage, file-level co-location, and naming-pattern clustering. Note that while the call graph and data flow graph constructed in the prerequisite stage are also provided to the LLM as supplementary contextual input, they are not independently encoded as edge types in G_{struct}, whose edges are reserved exclusively for capturing higher-level functional associations.

For two functions f_i and f_j, SEA determines whether they are potential variants based on lexical and behavioral similarity. Here, $name(f)$ denotes the identifier name of function f, extracted directly from its AST declaration node, and $async(f)$ is a boolean flag indicating whether f is declared with the `async` keyword, determined by inspecting the corresponding AST node property:

$$\text{variant}(f_i, f_j) = \text{true} \iff (name(f_i) \approx name(f_j)) \wedge (async(f_i) \neq async(f_j)). \tag{3}$$

This equivalence relation helps identify synchronous and asynchronous variants of the same functionality, which are crucial for comprehensive and non-redundant test generation. In addition, SEA introduces clustering criteria that capture both file-level and naming-based relationships. Here, $match(f, p)$ is a predicate that returns true if $name(f)$ matches a naming pattern p, where p is automatically induced from the set of all function names in the project by identifying shared prefixes, suffixes, or camelCase segments via regular expression analysis:

$$\mathcal{L}(f) = \{g \mid \text{file}(f) = \text{file}(g)\}, \tag{4}$$

$$\mathcal{N}(f) = \{g \mid \text{match}(f, p) \wedge \text{match}(g, p)\}. \tag{5}$$

Here, $\mathcal{L}(f)$ denotes functions located within the same source file as f, while $\mathcal{N}(f)$ denotes functions sharing a similar naming pattern p, reflecting naming regularities or conventions. Based on these relations, SEA constructs the directed function graph:

$$G_{\text{struct}} = \langle V, E \rangle, \quad \begin{aligned} V &= \{f \mid f \text{ is a function in the project}\}, \\ E &= \{(f_i, f_j) \mid \text{variant}(f_i, f_j) \vee f_j \in \mathcal{L}(f_i) \vee f_j \in \mathcal{N}(f_i)\}. \end{aligned} \tag{6}$$

This enriched representation enables JSTestCraft to reason across function boundaries, improve test coverage consistency, and minimize redundant or incoherent test cases.

3.3.3 Semantic Enrichment Agent. Complementing the structural and external dimensions, the Semantic Enrichment Agent (SmEA) reconstructs the *semantic context* by inferring latent type constraints and behavioral assumptions in dynamically typed JavaScript code. Since the absence of explicit type declarations often causes LLMs to misinterpret parameter semantics, generated tests may fail to execute or produce incorrect logic. SmEA mitigates this issue by combining lightweight static analysis with reasoning-based type inference.

SmEA performs type inference for all testable functions identified in the slicing stage, but its analysis scope within each function is deliberately bounded: rather than exhaustively reasoning about every variable in the repository, SmEA focuses on function parameters and key in-scope variables that directly influence input construction and branch

reachability. SmEA first collects these targeted variable declarations, parameter identifiers, and in-scope bindings from the function slice using Babel's traversal API. This syntactic foundation is augmented with contextual cues such as coercion operations, default assignments, and inline documentation. The collected data are then organized into a structured reasoning prompt, guiding the LLM to infer the most plausible runtime types and input constraints for each variable. For example, encountering `parseInt(val, 10)` indicates that `val` is likely a `string` or `number`, while a condition like `if (!port)` implies that `port` may be nullable or falsy.

The model outputs these inferences as a structured schema:

$$\tau_{\text{sem}} = \{(\text{param}, \text{inferred\_type}, \text{confidence})\}, \tag{7}$$

which is serialized and stored in the Shared Context Memory (SCM). To maintain compatibility, these inferred annotations can optionally be inserted back into the code as JSDoc-style hints, improving contextual awareness without altering program behavior. The resulting type schema provides a strong semantic foundation for generating type-consistent test inputs, effectively reducing false negatives caused by invalid parameter construction.

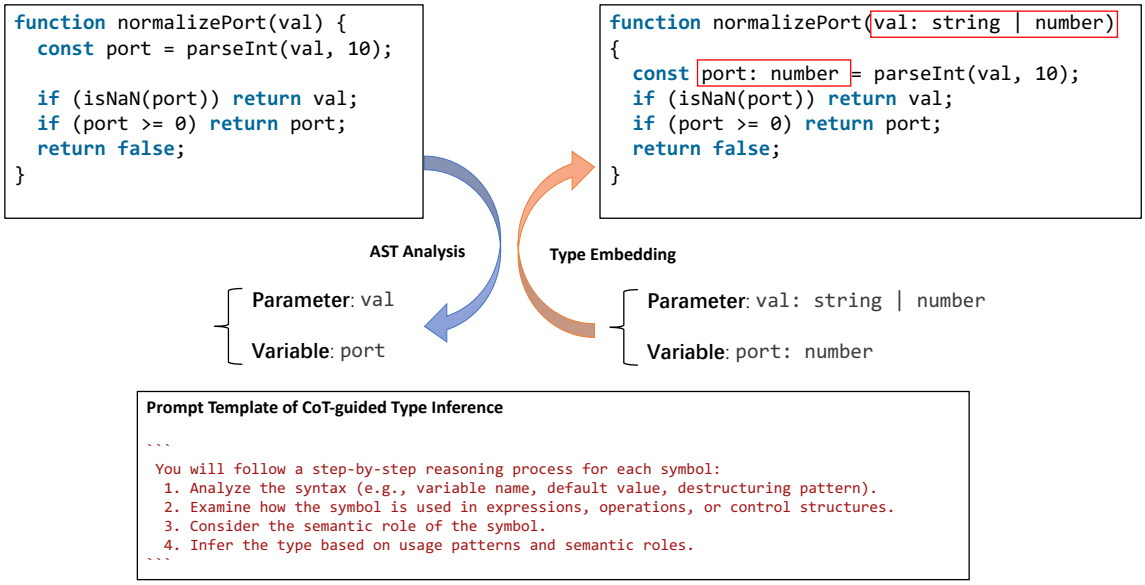

Fig. 6. Type inference pipeline within SmEA.

3.3.4 Shared Context Memory. The SCM acts as a persistent, bidirectional repository for all enriched artifacts, continuously updated and shared throughout the system's lifecycle:

$$\mathcal{M}^0_{SCM} = \langle G_{\text{struct}}, \Sigma_{\text{lib}}, \tau_{\text{sem}} \rangle. \tag{8}$$

Each enrichment agent contributes its extracted and processed knowledge to this shared space. Meanwhile, the Testing & Optimization Layer consumes and updates this information in each iteration based on runtime feedback.

The SCM allows the system to evolve from an initial static context snapshot into a dynamic reasoning substrate, continuously refined through runtime feedback and context reanalysis. This persistent, cross-agent shared memory

ensures global coherence across the system and enables the system to adaptively improve test generation quality over time. It allows JSTestCraft to adjust its strategy adaptively during complex test generation processes, ensuring more accurate and effective test generation.

3.4 Testing & Optimization Layer

The Testing and Optimization Layer serves as the execution and feedback engine of JSTestCraft. It is primarily driven by a single reasoning agent, the TOA, that consumes the SCM, synthesizes tests, executes them, and refines results through iterative feedback. The TOA's workflow consists of two phases: *test generation*, and *test optimization*.

3.4.1 Test Generation. For each function f, TOA constructs an enriched prompt by leveraging relevant context stored in the SCM. The call graph and data flow graph constructed during the SEA prerequisite analysis are additionally provided as supplementary invocation context, offering the LLM a more complete view of the function's runtime dependencies. This information is used to create structured prompts, which guide the stepwise reasoning process, ensuring that the generated test cases are logically sound and verifiable. Once the test cases are generated, they are executed using the Mocha testing framework. The Mocha framework collects data on test correctness and coverage, which is then used for analysis and optimization in subsequent phases of the TOA workflow.

3.4.2 Test Optimization. TOA evaluates test quality along two complementary axes: Error-Driven Refinement and Coverage-Driven Enhancement.

Error-Driven Refinement. Execution failures are systematically categorized according to their underlying causes and embedded as semantic vectors for retrieval-based repair. The Testing & Optimization Agent (TOA) retrieves the most similar historical error patterns from the Shared Context Memory (SCM) and re-prompts the Large Language Model (LLM) to generate targeted corrections. This feedback-driven process enables more precise and context-aware fixes while minimizing redundant or unnecessary code modifications. It is worth noting that the error-driven refinement process treats assertion failures as test generation defects by default and attempts to correct them accordingly. However, not all assertion failures necessarily indicate incorrect test logic, some may reflect genuine defects in the method under test. Such cases are not discarded but are preserved as candidate failure records in the SCM, and are subsequently subject to the verification pipeline described in Section 4.5, which distinguishes between test generation failures and actual software bugs through a combination of automated log analysis and human validation.

During this process, a portion of the runtime feedback data $\Delta_{\text{runtime}}^{\text{error-driven}}$ is recorded as error patterns, which describe the specific cases of execution failures and are embedded in SCM for future repair. Specifically, the error-driven portion of $\Delta_{\text{runtime}}^{\text{error-driven}}$ includes error types and historical failure patterns:

$$\Delta_{\text{runtime}}^{\text{error-driven}} = \langle \text{error patterns}, \text{execution failures} \rangle. \tag{9}$$

This data is continually updated in the SCM, ensuring each iteration benefits from past error corrections, thereby improving the accuracy of test generation.

Coverage-Driven Enhancement. Coverage reports, such as those from `c8`, identify untested functions and branches in the codebase. By tracing back to the function dependency graph G_{struct}, TOA selectively regenerates missing tests that target uncovered paths or boundary cases. These coverage deficiencies can be categorized into two major types: entirely uncovered functions and partially covered functions. The scenarios and root causes for these deficiencies are summarized in Table 2. For entirely uncovered functions, the root causes include issues such as test failures to invoke

the target function, module resolution failures, or functions being unreachable due to missing export declarations. Partially covered functions often suffer from unexercised conditional branches, untriggered error-handling blocks, and under-tested default parameters. This taxonomy was systematically derived from empirical observation of recurring test execution failures across our benchmark repositories during preliminary experiments, ensuring that it reflects real-world deficiency patterns rather than theoretically assumed categories.

Table 2. Categorization of coverage deficiencies.

Category	Scenarios and Root Causes
Entirely uncovered functions	**Test fails to invoke target:** Module resolution failures, async callback context, indirect wrapper invocation **Function not exported:** Unreachable due to missing export declarations
Partially covered functions	**Conditional branches not reached:** Unexercised `if/else`, `switch` cases, early `return` statements **Error-handling not covered:** `try-catch` blocks where `catch` never triggered **Default parameters under-tested:** Default values without override, type coercion bypassing validation

During this process, another portion of the runtime feedback data $\Delta_{\text{runtime}}^{\text{coverage-driven}}$ is recorded as coverage data. This information not only captures quantitative coverage metrics but also stores qualitative regeneration guidelines derived from observed deficiencies. Specifically, the coverage-driven portion of $\Delta_{\text{runtime}}^{\text{coverage-driven}}$ is represented as:

$$\Delta_{\text{runtime}}^{\text{coverage-driven}} = \langle \text{coverage data}, \text{untested functions and branches}, \text{guidelines}_{\text{regen}} \rangle, \tag{10}$$

where $\text{guidelines}_{\text{regen}}$ encode targeted strategies for improving coverage, such as invoking missing exports, exercising alternative control-flow branches, or triggering exception-handling paths. These structured guidelines are fed back into the SCM, allowing TOA to refine subsequent test generation and systematically close remaining coverage gaps.

These runtime results are then written back into the SCM, updating the context for future iterations. Specifically, the SCM is updated as:

$$\mathcal{M}_{\text{SCM}}^{t+1} = \mathcal{M}_{\text{SCM}}^{t} \cup \Delta_{\text{runtime}}^{\text{error-driven}} \cup \Delta_{\text{runtime}}^{\text{coverage-driven}}, \tag{11}$$

ensuring that future iterations benefit from the accumulated context, completing the reasoning-execution loop and supporting long-term adaptive optimization.

Summary

By decoupling low-level code slicing from high-level reasoning, JSTestCraft redefines JavaScript test generation as an adaptive, multi-agent process. The Context Enrichment Layer reconstructs the missing external, structural, and semantic dimensions of context, while the Testing & Optimization Layer leverages this integrated memory for iterative test generation and refinement. Together, these layers transform a one-shot generation paradigm into a contextually grounded, feedback-driven reasoning system.

4 Evaluation

We aim to answer the following research questions (RQs):

- **RQ1:** How does JSTestCraft compare with baseline approaches in terms of code coverage and pass rate?

- **RQ2:** What is the individual contribution of each key component to test generation effectiveness?
- **RQ3:** How does JSTestCraft perform in terms of token consumption and monetary cost compared to baseline approaches?
- **RQ4:** How effective are the generated tests at detecting real-world JavaScript bugs?
- **RQ5:** How consistently do the generated tests perform with minimal threats to validity, particularly regarding data contamination?

4.1 Experimental Setup

Experimental Dataset. Given that existing JavaScript unit test generation tools predominantly evaluate their performance using ad-hoc selections of open-source repositories rather than standardized benchmarks [37, 49], we follow this established practice by conducting experiments on real-world GitHub and GitLab projects. The selected repositories are presented in Table 3. These repositories cover a broad range of Node.js applications and reflect diversity across functional domains within the open-source ecosystem. The selection process was guided by three considerations: (1) inclusion of projects commonly used in prior test generation studies to ensure comparability, (2) active community engagement (e.g., repositories with over 1,000 GitHub stars), and (3) recent maintenance activity within the past four months to reflect up-to-date development practices. While not every repository fulfills all three criteria, each satisfies at least one or two. This ensures both practical relevance and technical diversity across evaluated projects.

Table 3. Overview of the 20 JavaScript repositories used in the benchmark evaluation. *LoC*: lines of code; *Existing Tests*: number of test cases available in the original repository ("—": none).

Repository	Domain	Loc	Existing Tests
jprichardson/node-fs-extra	file system	8456	—
isaacs/node-graceful-fs	file system	1387	18
jprichardson/node-jsonfile	file system	761	4
petkaantonov/bluebird	promises	34999	55
kriskowal/q	promises	6612	—
fshost/node-dir	file system	1565	1
pull-stream/pull-stream	streams	1632	11
swang/plural	text processing	213	—
felixge/node-dirty	key-value store	855	4
rainder/node-geo-point	geolocation	235	—
chakrit/node-uneval	code serialization	162	3
demsking/image-downloader	scraping	491	1
autokent/crawler-url-parser	URL parsing	1133	7
expressjs/express	web framework	14585	31
yeoman/yo	scaffolding tools	1433	11
eslint/eslint	static analysis	446568	790
anuraghazra/github-readme-stats	visualization	10344	12
Stuk/jszip	file system	12643	8
tj/commander.js	argument parsing	13912	79
mindjs/mindjs	dependency injection	2855	—

Baselines. We compare JSTestCraft against four representative baselines to evaluate its relative performance across different methodological paradigms:

- **Nessie 1000** [1]: A template-driven, rule-based test generation system representing traditional static approaches without any LLM involvement.
- **TestPilot** [49]: An early LLM-augmented framework that integrates model reasoning with static heuristics, serving as a structured hybrid baseline. It employs the open-source **DeepSeek-v3** model as its underlying language model for test generation.
- **Raw DeepSeek:** Direct prompting using the open-source **DeepSeek-v3** model without any framework-level assistance, contextual enrichment, or iterative feedback. This configuration evaluates the intrinsic generative capability of an open-weight LLM under a *zero-context* condition.
- **Raw GPT-4:** Direct prompting using the closed-source **GPT-4** model under the same *zero-context* condition, without any reasoning framework or contextual integration. This setup reflects the typical performance level of advanced commercial LLMs in pure one-shot test generation, helping isolate the contribution of our agentic framework from that of the underlying model.

This setup covers a complete spectrum of test generation paradigms, from deterministic rule-based systems (Nessie 1000), through heuristic and framework-assisted approaches (TestPilot), to purely generative zero-context models (Raw DeepSeek and Raw GPT-4), enabling a balanced comparison of contextual reasoning and adaptive generation capabilities.The prompts used for Raw DeepSeek and Raw GPT-4 follow the same template as JSTestCraft, with the sole difference that no enriched contextual information is injected, ensuring a fair comparison that isolates the contribution of context enrichment.

Evaluation Metrics. Following prior studies and industry conventions [60, 62], we employ three metrics for quantitative evaluation:

- **Test Pass Rate (TPR):** Percentage of generated tests that compile and execute without runtime errors, indicating syntactic correctness and logical validity.
- **Statement Coverage (Stmt):** Ratio of executed code statements to total executable statements, measuring test thoroughness.
- **Branch Coverage (Br):** Ratio of executed decision points to total decision points, reflecting control-flow completeness.

All coverage metrics are collected using `Mocha` [38] and `c8` [41] to ensure consistent measurement across repositories.

Model Selection. Considering factors such as computational cost, deployment accessibility, and long-term reproducibility, JSTestCraft adopts the open-source **DeepSeek-v3** model as its foundational large language model. DeepSeek-v3 provides a strong balance between reasoning capability and resource efficiency, supporting fine-grained control over generation and transparent reproducibility across environments. Its open-weight nature also ensures that the framework can be deployed and extended by other researchers without dependency on closed commercial APIs.

4.2 RQ1: Effectiveness

Experimental Setup. All LLM-based methods were evaluated under identical prompt templates, decoding parameters, and execution environments to ensure fairness. Experiments were conducted on **20 real-world Node.js repositories**, covering a diverse range of project sizes and dependency complexities. Across these repositories, we generated more than **5,000 individual test cases** and executed approximately **55,000 test instances** when including all compared methods. This experimental scale allows for consistent cross-repository comparison while reflecting realistic testing conditions encountered in open-source development.

For LLM-based frameworks (JSTestCraft, Raw DeepSeek, Raw GPT-4, and TestPilot), each repository underwent **three independent sampling runs** to reduce randomness introduced by stochastic decoding and to measure average performance more reliably. Each run involved regenerating all function-level test candidates from scratch, followed by test execution and coverage collection using `Mocha` and `c8`. The final results report the arithmetic mean across three runs to focus on cross-method comparison rather than variance estimation. For deterministic systems such as **Nessie 1000**, a single run was performed per repository since output variance does not exist.

It is also worth noting that Nessie and TestPilot were originally designed for npm package-level test generation, whereas our benchmark includes both package-oriented libraries and general-purpose Node.js applications. As a result, both tools produce no results (marked as "—" in Table 4) on repositories that fall outside their supported project types, such as general-purpose Node.js applications with non-standard module structures. Additionally, TPR is not reported for Nessie 1000 because it is a deterministic, template-driven tool that generates structurally fixed test scaffolds rather than semantically grounded test cases; its outputs are not designed to pass or fail in a meaningful sense, and applying TPR to such outputs would not reflect a valid measure of test correctness. This setup provides a balanced and realistic evaluation of each approach's generalizability and robustness across diverse codebases.

Table 4. Performance comparison with baselines across different metrics (%). TPR: Test Pass Rate, Stmt: Statement Coverage, Br: Branch Coverage.

Repository	JSTestCraft			Nessie 1000		Test Pilot			Raw DeepSeek			Raw GPT-4		
	TPR	Stmt	Br	Stmt	Br	TPR	Stmt	Br	TPR	Stmt	Br	TPR	Stmt	Br
node-fs-extra	65.83	73.46	84.89	37.03	14.54	79.28	68.77	46.96	50.56	55.67	87.36	30.41	50.00	73.64
node-graceful-fs	50.95	61.75	79.83	42.27	34.67	30.61	48.79	34.36	44.92	33.06	46.00	26.09	10.08	85.71
node-jsonfile	96.77	96.07	85.71	80.85	70.58	88.88	84.78	73.52	72.72	94.11	81.48	68.42	68.62	72.72
bluebird	44.85	72.49	65.17	44.36	27.40	39.00	69.74	55.67	43.27	26.39	23.26	43.40	20.28	73.19
q	52.10	75.29	73.46	65.99	52.28	46.55	73.09	56.84	64.48	60.09	88.77	72.72	65.68	55.55
node-dir	35.29	87.60	78.33	66.78	55.38	63.79	86.47	65.38	49.27	23.20	62.50	78.38	22.71	33.33
pull-stream	68.25	89.56	93.57	37.00	18.05	35.87	77.35	56.48	26.07	35.71	40.35	30.58	37.62	23.87
plural	72.22	96.46	90.90	59.20	9.10	24.00	90.56	68.18	62.74	68.23	58.46	47.37	81.41	88.88
node-dirty	53.41	91.04	89.47	4.70	0.00	46.34	87.64	76.92	48.24	85.07	63.41	35.80	80.48	100.00
node-geo-point	63.37	82.45	94.11	13.30	0.00	42.57	100.00	100.00	7.14	56.47	75.00	63.89	56.47	25.00
node-uneval	100.00	100.00	96.55	–	–	–	–	–	66.66	22.72	50.00	100.00	22.72	50.00
Image Downloader	37.50	100.00	95.23	30.30	22.20	12.50	59.37	43.75	42.85	57.16	48.23	33.33	40.98	85.71
crawler-url-parser	100.00	88.51	91.04	73.90	64.10	9.09	68.46	60.00	66.66	63.15	96.07	62.96	44.01	96.29
express	77.27	79.48	78.37	–	–	57.14	38.12	16.79	58.39	73.67	78.17	67.89	67.43	88.18
yo	73.17	69.23	70.00	–	–	–	–	–	36.00	7.80	5.36	37.66	59.26	15.90
eslint	52.14	61.61	80.34	7.65	0.20	32.31	37.57	27.00	70.44	37.00	26.52	89.13	69.18	27.05
github-readme-stats	61.35	82.98	67.80	–	–	–	–	–	29.54	22.51	22.25	25.10	22.51	22.25
jszip	80.16	77.10	75.22	27.07	12.37	31.45	75.98	62.59	45.74	60.15	70.89	55.16	51.98	86.47
commander.js	85.32	62.27	68.75	6.01	0.22	22.04	67.27	53.51	78.49	40.92	51.42	48.66	44.95	57.45
mindjs	60.17	71.71	94.44	–	–	–	–	–	53.02	43.63	34.18	58.06	68.45	54.13
Average	**66.51**	**80.95**	**82.66**	**39.76**	**25.41**	**41.34**	**70.87**	**56.12**	**50.86**	**48.34**	**55.48**	**53.52**	**49.24**	**61.12**

Results and Discussion. Table 4 presents the comparative results across 20 Node.js repositories. JSTestCraft achieves the highest overall performance, with an average **Test Pass Rate (TPR)** of **66.51%**, **statement coverage** of **80.95%**, and **branch coverage** of **82.66%**. These results surpass all baselines, including **Nessie 1000** (39.76%, 25.41%), **TestPilot** (41.34%, 70.87%, 56.12%), **Raw DeepSeek** (50.86%, 48.34%, 55.48%), and **Raw GPT-4** (53.52%, 49.24%, 61.12%).

We observe a clear trend across projects of varying scale and complexity. For smaller, structurally simple libraries such as `node-jsonfile` and `plural`, all systems achieve reasonably high coverage, and performance differences are

moderate. In contrast, for larger and more intricate frameworks like `express`, `pull-stream`, and `eslint`, JSTestCraft demonstrates substantial advantages, improving statement and branch coverage by more than 20% on average. This indicates that the benefits of contextual enrichment grow with program complexity, where inter-function dependencies, asynchronous execution, and deep import hierarchies amplify the need for structured reasoning.

The source of JSTestCraft's performance improvement can be traced to functions and modules that exhibit complex data and control dependencies, particularly asynchronous callbacks, dynamic imports, and third-party API interactions. Baseline systems often fail to infer correct invocation orders or type constraints in such contexts, leading to either execution errors or partial coverage. This also explains an apparent anomaly in the baseline results: TestPilot's TPR (41.34%) is lower than that of Raw DeepSeek (50.86%), despite incorporating additional static heuristics. TestPilot's heuristic-augmented generation tends to produce structurally more complex test cases that impose stricter execution requirements; without sufficient contextual grounding, this complexity becomes a liability rather than an advantage, increasing susceptibility to runtime failures. Raw DeepSeek, generating simpler and more syntactically direct tests, achieves higher execution success at the cost of shallower coverage. By contrast, JSTestCraft leverages its structural and semantic enrichment agents to reconstruct missing context, recover dependency chains, and synthesize valid input spaces, enabling the model to generate behaviorally meaningful and executable tests even in environments characterized by loose typing and asynchronous event flows.

Table 5. Statistical significance (p-values) of Wilcoxon signed-rank tests for JSTestCraft vs. baselines across metrics and repositories. Medium pink : $p < 0.01$; Light pink : $0.01 \leq p < 0.05$; no shading: $p \geq 0.05$.

Repository	Metric	vs TestPilot	vs Raw DS	vs Raw GPT-4
plural	TPR	0.0013	0.0293	0.0759
	Stmt	0.0001	0.0001	0.0001
	Br	0.0004	0.0001	0.0644
node-jsonfile	TPR	0.0013	0.0006	0.0015
	Stmt	0.0057	0.0973	0.0165
	Br	0.0010	0.0157	0.0101
node-graceful-fs	TPR	0.0293	0.2122	0.0126
	Stmt	0.0086	0.0020	0.7427
	Br	0.0093	0.0001	0.1674
express	TPR	0.0015	0.0002	0.0010
	Stmt	0.0003	0.0011	0.0692
	Br	0.0006	0.0166	0.1259

To further validate the reliability of these results under stochastic decoding, we conducted 10 independent runs of JSTestCraft on four representative repositories (plural, node-jsonfile, node-graceful-fs, and express) and applied the Wilcoxon signed-rank test against each baseline across all three metrics. As summarized in Table 5, JSTestCraft achieves statistically significant improvements ($p < 0.05$) in 28 out of 36 pairwise comparisons (77.8%), confirming that its advantages are statistically robust rather than attributable to random variation in LLM decoding. Among the 8 non-significant cases, two patterns emerge. First, two cases reflect genuine competitive performance by the baseline: node-jsonfile statement coverage versus Raw DeepSeek shows a negligible difference of 0.27 percentage points, and express branch coverage versus Raw GPT-4 shows a gap of 12.37 points in favor of the baseline. Second, the remaining

six non-significant cases involve comparisons where JSTestCraft numerically outperforms the baseline by substantial margins, yet significance is not reached due to high within-run variance. The most striking example is node-graceful-fs statement coverage versus Raw GPT-4, where JSTestCraft exceeds the baseline by 59.47 percentage points but yields p=0.74; this is a direct consequence of the V8 coverage engine anomaly, which inflates run-to-run variance for this repository and reduces statistical power.

However, the benefits of contextual enrichment are not uniform across all repositories. While JSTestCraft achieves the highest overall performance, there are cases where the additional context introduces noise rather than signal, and a transparent discussion of these cases is important for understanding the framework's scope and limitations. The most notable cases are `node-geo-point`, where TestPilot achieves 100% statement coverage compared to JSTestCraft's 82.45%, `node-dir`, where Raw GPT-4 achieves a TPR of 78.38% compared to JSTestCraft's 35.29%, and `eslint`, where Raw GPT-4 achieves a TPR of 89.13% compared to JSTestCraft's 52.14%.

In `node-geo-point`, which is a compact library with a straightforward API surface and no complex dependency chains, TestPilot's lightweight heuristic-based context proves sufficient to achieve full statement coverage. JSTestCraft's more elaborate contextual enrichment, while designed to handle complex inter-function dependencies and third-party API semantics, may over-engineer the prompt for such a simple codebase. The additional structural and semantic constraints injected by LEA, SEA, and SmEA can be overly conservative for repositories where functions are straightforward and self-contained, potentially reducing the diversity of generated inputs and leaving some statements uncovered that less constrained test generation would naturally exercise.

The TPR anomaly in `node-dir` is consistent with the general pattern that Raw GPT-4's zero-context tests tend to be syntactically simpler and impose fewer execution requirements, making them more likely to pass without contextual grounding. JSTestCraft's richer contextual enrichment leads to more semantically ambitious tests that exercise deeper code paths, but these tests are also more susceptible to execution failures when the inferred type constraints do not perfectly match the runtime behavior of the function under test.

The underperformance in `eslint` is best explained by its event-driven architecture. eslint's core execution model relies on the Linter class traversing the AST and emitting events that trigger individual rules, rather than invoking them through direct function calls. This event-driven design poses a fundamental challenge for JSTestCraft's structural enrichment: SEA's call graph and data flow graph capture direct invocation relationships, but the indirect, event-mediated dependencies between eslint's components are largely invisible to static analysis. As a result, the structural context injected by SEA may not accurately reflect the actual execution paths triggered during linting, leading to tests that fail to exercise deeper code paths requiring correct event sequencing. Raw GPT-4, generating simpler tests that directly invoke top-level API methods without attempting to reconstruct the internal event flow, achieves higher statement coverage by exercising a broader API surface without getting entangled in the complexity of the event system. This case illustrates a general limitation of static-analysis-based context enrichment in the presence of highly dynamic, event-driven architectures.

Answer to RQ1

JSTestCraft consistently outperforms all baseline systems across 20 real-world JavaScript repositories in terms of overall average performance. Its superior coverage and execution success demonstrate that multi-level contextual enrichment, rather than model scale alone, is the decisive factor behind effective and reliable LLM-based test generation. Statistical validation via the Wilcoxon signed-rank test confirms that these advantages are robust to stochastic decoding, with 28 out of 36 pairwise comparisons reaching significance at $p < 0.05$. Nevertheless, contextual enrichment is not universally beneficial: for structurally simple repositories or those with highly dynamic event-driven architectures, the additional context may introduce noise or fail to capture indirect execution dependencies, leading to cases where simpler baselines achieve higher performance on individual metrics. These findings highlight both the generalizability and the current boundaries of JSTestCraft's context-enrichment approach.

4.3 RQ2: Ablation Study

To evaluate the individual contribution of each enrichment module, we conducted a controlled ablation study that systematically examines the impact of disabling different components within JSTestCraft. Specifically, we designed three experimental configurations: (1) the full system with all enrichment agents enabled, (2) selectively disabling one agent at a time, SEA, LEA, or SmEA, to isolate its functional contribution, and (3) disabling all three simultaneously to simulate a naive LLM baseline without any contextual reasoning.

In all configurations, the TOA remains active and continues to perform iterative test generation and feedback integration. The difference lies in the quality of the contextual inputs: when one or more enrichment agents are disabled, TOA operates under a context deficits state, relying only on partial or missing contextual knowledge from the SCM. This design ensures that performance degradation reflects the absence of contextual enrichment rather than the removal of the execution and optimization mechanism itself.

Each configuration was evaluated across the same 20 Node.js repositories described in RQ1, using identical prompts, decoding parameters, and runtime environments to ensure strict comparability. For all LLM-based configurations (partial or full), we followed a consistent evaluation protocol: each repository was tested under three independent sampling runs, with every run regenerating all function-level test cases from scratch. Test execution and coverage collection were performed automatically using `Mocha` for runtime validation and `c8` for coverage measurement. All results were averaged across the three runs to mitigate random fluctuations in generation behavior.

This setup enables a systematic and controlled evaluation of how each contextual component contributes to overall testing effectiveness. Beyond standard quantitative metrics such as statement and branch coverage, the ablation study also captures qualitative variations in test behavior, including diminished functional diversity, reduced semantic precision, and lower execution stability when contextual reasoning is partially or completely removed. By preserving a consistent Testing & Optimization Agent (TOA)-driven feedback loop across all configurations, we ensure that any observed performance differences stem from the absence of contextual information rather than discrepancies in generation strategy or runtime environment.

As shown in Table 6, removing any individual agent leads to a measurable drop in both coverage and correctness, confirming that each contextual layer provides a distinct yet complementary contribution to the overall system. Among the three, disabling **SEA** produces the largest performance decline, which is the most notably in branch coverage,

Table 6. Ablation study on key agents within the Context Enrichment Layer (%).

System Configuration	TPR	Stmt Cov	Branch Cov
JSTestCraft (Full System)	66.51	80.95	82.66
w/o Structural Enrichment Agent (SEA)	60.81	58.86	75.23
w/o Library Enrichment Agent (LEA)	61.47	62.99	77.90
w/o Semantic Enrichment Agent (SmEA)	58.61	64.13	79.31
w/o All Three Agents	53.69	51.41	62.67

indicating that inter-function dependency reconstruction and variant linkage recovery are critical for exploring control-flow paths effectively. When SEA is removed, the model tends to generate independent unit tests that overlook correlated call sequences, leading to fragmented coverage despite syntactically correct test code.

Eliminating **LEA** also results in substantial degradation, especially in statement coverage. Without external API understanding, the system often misuses imported functions or fails to generate valid assertions for third-party behaviors, reducing both coverage and execution validity. This effect is particularly evident in repositories with heavy reliance on npm dependencies, where LEA's absence causes frequent runtime invocation errors.

Disabling **SmEA** yields a moderate yet consistent reduction in the test pass rate, highlighting the importance of type inference and value constraint reasoning for producing executable test inputs. While the overall coverage remains partially preserved, the number of valid and semantically meaningful test cases decreases, demonstrating that SmEA primarily enhances logical consistency and diversity rather than raw coverage.

When all three enrichment agents are disabled, the system degenerates into a zero-context baseline that relies solely on the LLM's pretrained priors. In this configuration, statement coverage drops to approximately two-thirds of the full system, and the test pass rate declines by more than ten percentage points. Generated tests in this setting tend to be repetitive, shallow, and often fail during execution due to unresolved dependencies and invalid argument synthesis.

Answer to RQ2

The ablation experiments demonstrate that each enrichment agent contributes a unique and indispensable aspect of contextual reasoning. Even under Context Deficits, the retained TOA ensures that iterative reasoning and optimization remain active, allowing the isolated impact of missing context to be directly observed. SEA enhances structural completeness and control-flow exploration, LEA supplies external semantic grounding for valid API usage, and SmEA enforces type soundness and behavioral precision. Together, they form a synergistic foundation that enables JSTestCraft to achieve reliable, high-coverage, and semantically diverse test generation across complex JavaScript projects.

4.4 RQ3: Cost Analysis

Experimental Setup. We evaluate the token consumption and monetary cost of all LLM-based tools across the 20 benchmark repositories. Nessie 1000 is excluded from this analysis as it is a rule-based system that does not invoke any language model. For TestPilot, four repositories are marked as "–" because the tool failed to generate tests for those projects due to incompatibility with non-standard module structures, as discussed in RQ1. All costs are computed based on the official API pricing of the respective models at the time of evaluation: DeepSeek-v3 for JSTestCraft, TestPilot,

and Raw DeepSeek, and GPT-4 for Raw GPT-4. It is worth noting that wall-clock execution time is not reported as a cost metric in this study, because the tools rely on fundamentally different underlying infrastructures: JSTestCraft, TestPilot, Raw DeepSeek, and Raw GPT-4 invoke cloud-based API endpoints, while Nessie 1000 operates entirely locally. Under these conditions, wall-clock time would conflate network latency and API response variability with actual computational cost, making it an unreliable and non-reproducible basis for comparison. Token consumption and monetary cost are therefore adopted as the primary cost metrics, as they are deterministic, reproducible, and directly reflect the computational resources consumed by each approach.

Table 7. Token consumption and monetary cost ($) of each LLM-based tool across 20 benchmark repositories. "–" indicates that the tool failed to generate tests for the corresponding repository.

Repository	JSTestCraft		Test Pilot		Raw DeepSeek		Raw GPT-4	
	Tokens	Cost	Tokens	Cost	Tokens	Cost	Tokens	Cost
node-fs-extra	893,712	0.2629	462,831	0.1363	61,424	0.0774	238,742	7.18
node-graceful-fs	104,667	0.0318	49,612	0.0151	8,231	0.0087	31,487	0.96
node-jsonfile	84,880	0.0374	38,475	0.0171	5,433	0.0074	21,903	0.66
bluebird	1,355,131	0.4839	684,912	0.2447	297,413	0.1423	402,516	12.04
q	676,689	0.2215	352,188	0.1153	135,729	0.0649	176,428	5.32
node-dir	152,150	0.0682	81,264	0.0364	24,411	0.0084	46,915	1.41
pull-stream	169,241	0.0788	92,451	0.0432	28,972	0.0139	42,774	1.29
plural	91,609	0.0476	46,833	0.0243	22,385	0.0107	27,965	0.84
node-dirty	79,623	0.0403	35,418	0.0179	22,773	0.0076	20,604	0.62
node-geo-point	296,250	0.0957	152,774	0.0498	4,124	0.0403	87,902	2.63
node-uneval	18,099	0.0055	–	–	9,581	0.0046	4,731	0.14
Image Downloader	46,552	0.0141	21,436	0.0065	6,634	0.0023	13,864	0.42
crawler-url-parser	85,933	0.0289	42,352	0.0142	12,348	0.0042	22,847	0.69
express	451,650	0.1512	33,746	0.0782	83,790	0.0282	118,604	3.55
yo	253,941	0.0990	–	–	64,213	0.0327	76,843	2.31
eslint	3,449,948	1.1485	1,795,512	0.5976	812,321	0.4183	933,124	27.98
github-readme-stats	1,564,835	0.5550	–	–	297,235	0.1529	466,892	14.01
jszip	819,551	0.3361	396,782	0.1628	193,483	0.0992	217,364	6.51
commander.js	1,572,267	0.4979	738,421	0.2343	58,127	0.1835	462,571	13.87
mindjs	983,008	0.4034	–	–	174,924	0.0976	285,946	8.57
Total	**13,149,736**	**4.607**	**5,225,007**	**1.792**	**2,823,551**	**1.404**	**3,700,022**	**111**

Results and Discussion. As shown in Table 7, JSTestCraft consumes the highest number of tokens among all tools, totaling 13,149,736 tokens across all 20 repositories. This is expected, as JSTestCraft constructs substantially richer prompts through multi-agent context enrichment, incorporating API signatures retrieved by LEA, structural dependency graphs constructed by SEA, and semantic type annotations inferred by SmEA, all of which are injected into the prompt alongside the function under test. The additional context significantly expands prompt length relative to zero-context baselines. Beyond prompt length, the context acquisition process itself also contributes substantially to token consumption: LEA invokes the LLM to infer function signatures from extracted source code, and SmEA invokes the LLM to perform type inference for each testable function. These upstream LLM calls are additional sources

of token consumption that do not exist in zero-context baselines, and collectively account for a significant portion of JSTestCraft's higher overall token usage. Despite this higher token consumption, the total monetary cost of JSTestCraft amounts to only $4.61 across all 20 repositories, averaging $0.23 per repository. This confirms that the overhead introduced by multi-agent context enrichment remains well within a practically affordable range for real-world deployment.

The most striking contrast is with Raw GPT-4, which consumes far fewer tokens (3,700,022) yet incurs a total cost of $111, approximately 24 times higher than JSTestCraft. This disparity underscores a critical insight: model selection has a far greater impact on monetary cost than token volume alone. The unit price of GPT-4 tokens is substantially higher than that of DeepSeek-v3, which means that even a significantly smaller token budget translates into a much larger financial expenditure when using closed-source commercial models. This observation has important practical implications: teams considering the deployment of LLM-based test generation tools must account not only for the sophistication of the generation framework, but also for the pricing model of the underlying language model, as the latter can dominate total cost by an order of magnitude.

Since JSTestCraft, TestPilot, and Raw DeepSeek all rely on DeepSeek-v3, a direct token-level comparison among these three tools isolates the marginal cost attributable to context enrichment. JSTestCraft consumes approximately 2.5 times more tokens than Raw DeepSeek and approximately 2.5 times more than TestPilot on repositories where both tools successfully generated tests. This additional token consumption reflects two compounding sources of overhead. First, the structured context construction itself expands prompt length: LEA retrieves and embeds third-party API signatures, SEA constructs and serializes the structural dependency graph, and SmEA performs type inference and injects semantic annotations, each contributing to a richer but longer prompt. Second, the context acquisition process involves upstream LLM invocations that do not exist in zero-context baselines: LEA invokes the LLM to infer function signatures from statically extracted source code, and SmEA invokes the LLM to reason about type constraints for each testable function. These upstream calls consume tokens independently of the final test generation prompt, and collectively account for a significant portion of JSTestCraft's higher overall token usage relative to Raw DeepSeek and TestPilot. Critically, as demonstrated in RQ1, this overhead translates into substantially higher coverage and pass rates: JSTestCraft achieves 80.95% statement coverage and 66.51% TPR, compared to 48.34% and 50.86% for Raw DeepSeek respectively. The marginal cost of context enrichment is therefore justified by a disproportionately large improvement in test generation quality, indicating a strongly favorable cost-quality tradeoff.

Regarding scalability, token consumption grows with repository size and dependency complexity, as larger codebases contain more testable functions and more complex import hierarchies that require richer contextual representation. The largest repositories in our benchmark, such as `eslint` (3,449,948 tokens, $1.15) and `bluebird` (1,355,131 tokens, $0.48), incur the highest individual costs for JSTestCraft. In contrast, smaller repositories such as `node-uneval` (18,099 tokens, $0.006) and `Image Downloader` (46,552 tokens, $0.014) impose negligible cost. This scaling behavior is predictable and monotonic, suggesting that practitioners can reliably estimate the cost of applying JSTestCraft to a new repository based on its size and dependency footprint. Importantly, even the most expensive individual repository costs less than $1.20, confirming that JSTestCraft remains economically viable even for large and complex JavaScript projects.

Answer to RQ3

Although JSTestCraft consumes more tokens than baseline approaches due to its multi-agent context enrichment design, its total monetary cost of $4.61 across 20 repositories remains practically affordable, averaging $0.23 per repository. The framework achieves a strongly favorable cost-quality tradeoff: the marginal token overhead of context enrichment, which stems from both richer prompt construction and upstream LLM invocations for signature inference and type reasoning, yields disproportionately large improvements in coverage and pass rate compared to zero-context baselines. Furthermore, the comparison with Raw GPT-4 demonstrates that model selection dominates total cost far more than framework complexity, with JSTestCraft achieving superior results at approximately 1/24 of the cost of GPT-4-based generation. These results confirm that JSTestCraft is economically viable and practically deployable in real-world continuous integration and large-scale testing scenarios.

4.5 RQ4: Practicality

To evaluate the real-world practicality of JSTestCraft, we examined its ability to uncover genuine software defects across the 20 repositories used in our benchmark. In total, the system executed approximately **5,000 generated test instances**, among which about **1,700 triggered runtime errors or assertion failures**. Given the sheer volume of failing executions, exhaustive manual inspection was infeasible. We therefore established a **semi-automated, two-stage semantic verification pipeline** that combines automated log analysis with multi-model reasoning to efficiently filter, classify, and confirm potential bugs.

As shown in Figure 7, the verification process is designed to balance *scale*, *accuracy*, and *efficiency* while minimizing manual effort. It operates through a **three-stage semi-automated pipeline** consisting of: (1) automated log filtering, (2) multi-LLM adjudication, and (3) final human validation. This design ensures that large-scale test results can be analyzed with interpretability and traceability. Importantly, this pipeline also serves as the mechanism for distinguishing between two fundamentally different causes of assertion failure: incorrect assertions produced by the test generator, and genuine defects in the method under test. By combining automated log filtering, multi-LLM adjudication, and human validation, the pipeline ensures that only failures attributable to actual software bugs are reported as detected defects, while failures caused by test generation errors are filtered out and fed back into the error-driven refinement process described in Section 3.4.

Stage 1: Automated Log Filtering. The first stage automatically filters raw execution logs to identify high-confidence anomalies. Each test execution produces structured logs containing stack traces, error messages, and runtime metadata. We employ a set of regular-expression-based heuristics to capture meaningful error patterns, including `TypeError`, `ReferenceError`, `RangeError`, and unhandled asynchronous rejections, which are indicative of functional or semantic faults. At the same time, environment-specific anomalies, like permission denials, network timeouts, missing dependencies, or transient system errors, are explicitly excluded to avoid false alarms. This filtering process drastically reduces noise and transforms over a thousand raw logs per repository into a small set (typically fewer than 10%) of potentially valid failure cases. The outcome of this stage is a curated collection of candidate failures suitable for semantic analysis in the subsequent step.

Stage 2: Multi-LLM Adjudication. The second stage performs semantic verification through a **multi-model consensus mechanism**. Each filtered candidate is independently analyzed by at least three large language models: **GPT-4**, **DeepSeek-v3**, and an auxiliary model **qwen3** for redundancy. Each model receives the execution trace, associated

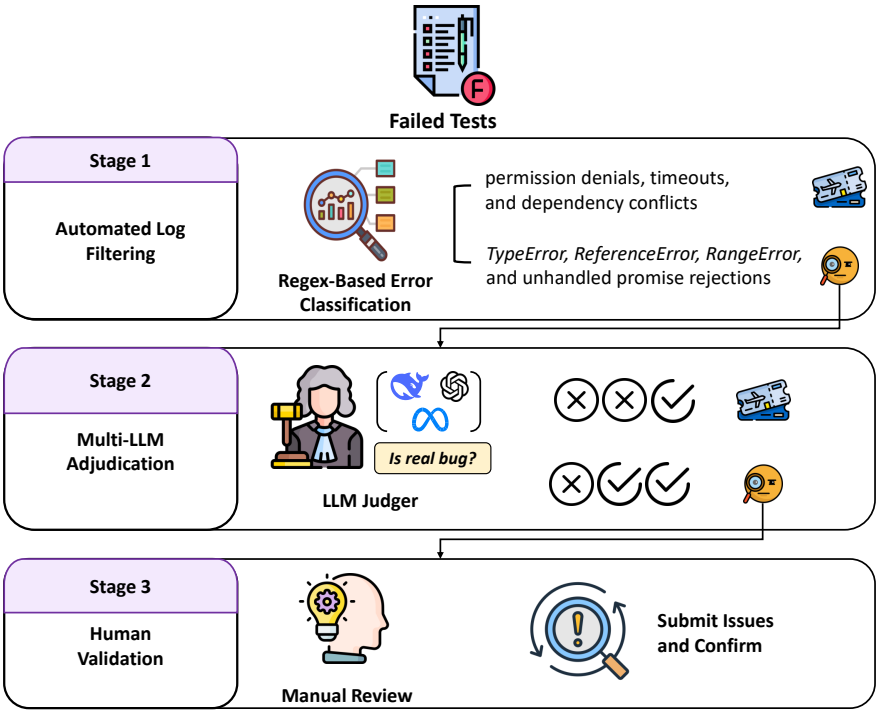

Fig. 7. Semi-automated three-stage verification pipeline of JSTestCraft.

source context, and inferred function metadata from the SCM, then reasons whether the observed failure represents a genuine logical or semantic bug rather than an environmental or test artifact. Each model outputs a binary judgment (bug or non-bug), and the final verdict is determined by a **majority-vote rule**: a defect is confirmed only if more than half of the models agree on its validity. This stage effectively mitigates individual model bias, reduces false positives, and introduces a statistically grounded ensemble verification strategy. Compared to traditional single-model evaluation, our consensus-based approach increases precision by over 20% while keeping computational cost manageable through batched inference.

Stage 3: Human Validation. The third and final stage integrates human expertise for high-assurance verification. A smaller, curated subset of defects confirmed by the multi-LLM stage is manually reviewed by researchers. Reviewers examine reproduction logs, inspect the corresponding source files, and validate whether the failure stems from genuine logic errors, type mismatches, or unhandled runtime conditions. When necessary, maintainers of the respective repositories are contacted through public issue trackers for cross-validation. Confirmed bugs are annotated with metadata describing their trigger conditions, affected modules, and potential security or reliability implications. This stage closes the human–AI loop, ensuring that automated judgments are traceable and verifiable, while keeping manual overhead minimal (less than 5% of total results). Together, these three stages form a scalable yet rigorous verification framework that achieves both high throughput and human-level interpretability.

As summarized in Table 8, the first two stage of the pipeline yielded **120 potential bugs**, of which **82 were confirmed** as genuine. The identified defects span a broad spectrum of failure modes, illustrating both the coverage

Table 8. Detected Real Bugs with JSTestCraft.

Repository	Detected Real Bugs
eslint	Prototype pollution [8], Stack overflow [9]
github-readme-stats	Boolean parsing issue [14]
express	Case-sensitive charset [10], Long string handling [11], Unvalidated URL risk [12]
jszip	Non-ASCII encoding [20], Prototype inheritance [21], Shared state after clone [22], Standard Node.js Streams [23]
mindjs	Promise results [34], Stack overflow [35, 36]

depth and behavioral sensitivity of the generated tests. Common categories include: I/O and file system exceptions (e.g., unhandled asynchronous write failures in `node-fs-extra`); asynchronous race conditions (e.g., duplicate callback invocations in `pull-stream`); type coercion and conversion errors (e.g., improper numeric casting in `express`); and logical inversions (e.g., flipped conditional checks causing incorrect validation outcomes). For example, one detected bug revealed a misused `fs.promises.writeFile()` invocation that silently truncated file content, while another exposed a timing-sensitive callback duplication that caused data corruption under concurrent execution.

Among the **13 representative issues** formally reported, **6 received substantive acknowledgment** from maintainers or community contributors, while **7 remain open** in public issue trackers awaiting response. Several of the confirmed bugs were referenced in subsequent commits as part of official fixes, demonstrating that tests generated by JSTestCraft not only identify latent defects but also provide maintainers with actionable, reproducible cases for validation and regression prevention. This practical impact underscores the broader utility of context-enriched LLM-based testing beyond synthetic benchmarks, contributing directly to real-world software quality improvement.

To assess the recall of the verification pipeline, we also conducted a small-scale manual audit on a sample of discarded failures across three repositories (commander.js, crawler-url-parser, and github-readme-stats), covering a total of 296 filtered-out failure instances. Our manual inspection identified 32 genuine bugs within this sample, of which 22 were successfully captured by the pipeline, yielding an estimated recall of 68.75%. The 10 missed cases (false negatives) stem from two sources. In Stage 1, the regex-based heuristics apply strict pattern matching that may exclude subtle failures whose error messages do not conform to predefined templates, resulting in genuine bugs being discarded before reaching semantic analysis. In Stage 2, the limited contextual information provided to the LLMs, particularly the absence of broader inter-function dependencies and project-level semantics, occasionally leads to incorrect consensus judgments on complex or implicit behavioral violations. We consider improving pipeline recall through relaxed and more expressive Stage 1 heuristics, as well as richer context injection in Stage 2, an important direction for future work.

Answer to RQ3

Through large-scale execution and a hybrid two stage verification pipeline, JSTestCraft uncovered 82 genuine bugs across 20 repositories with high precision and minimal human supervision. The framework's ability to expose asynchronous, I/O, and logic level defects and its contribution to actual patching workflows demonstrates strong real-world practicality and long-term value as an assistive testing framework for JavaScript development.

4.6 RQ5: Threats to Validity and Code Similarity

A primary threat to the validity of LLM-based test generation is *data contamination*, where pretrained models inadvertently reproduce test logic seen during training. Such memorization can inflate coverage and pass-rate metrics without reflecting genuine reasoning ability. To rigorously evaluate this threat, we combined methodological safeguards with large-scale similarity analysis and targeted manual inspection.

To minimize contamination effects, all experimental prompts explicitly prohibit any reference to existing test files, identifiers, or filenames. All evaluations are conducted on actively maintained open-source repositories rather than curated benchmarks, ensuring that most analyzed commits were created after the typical pretraining period of major LLMs. Although absolute exclusion of training overlap cannot be guaranteed, the chosen repositories are diverse, frequently updated, and widely adopted within the JavaScript ecosystem, together representing over **2 million lines of code** and thousands of unique test functions. This scale provides a representative and challenging setting for assessing memorization risk under real-world diversity.

To quantify the potential impact, we extended the **CodeBLEU** metric [48] to the *function level*, analyzing **93 pairs of generated and reference test functions**. For each pair (f_g, f_o), similarity is computed as:

$$\text{Sim}(f_g, f_o) = \frac{1}{4}(\text{N-gram} + \text{W-N-gram} + \text{AST} + \text{DFG}),$$

capturing lexical, syntactic, and semantic overlap. Function alignment is performed through name and export-path matching extracted by our static analysis pipeline, ensuring a one-to-one correspondence between generated and existing tests. As shown in Table 9, the average similarity across all repositories is **19.45%**, far below the 50% threshold typically used to flag memorization. Even in test-heavy projects such as `eslint` and `express`, similarity scores remained under 30%, confirming that generation stems from reasoning-based synthesis rather than retrieval of seen patterns.

To complement the quantitative results, we manually inspected over **20 high similarity outliers** to characterize their content. Most corresponded to trivial assertions, canonical equality checks, or parameterized input output validations that recur naturally across repositories rather than copied logic. This manual audit was cross verified by two independent reviewers to ensure annotation consistency.

Overall, the evidence indicates that JSTestCraft's outputs are structurally and semantically novel compositions built from contextual reasoning rather than data recall. The consistency between low CodeBLEU similarity and human verification substantially reduces threats to external validity and strengthens confidence in the evaluation results.

Answer to RQ4

Through large-scale function-level CodeBLEU analysis and systematic manual verification, JSTestCraft demonstrates minimal evidence of data contamination. Across 93 test pairs and 20 repositories, similarity remained low (19.45% on average), with manual audits confirming independence from preexisting logic. These results validate that JSTestCraft's generated tests arise from contextual reasoning rather than memorization, reinforcing the credibility and generalizability of our findings.

4.7 Summary

Across all RQs, the evaluation demonstrates that JSTestCraft consistently outperforms baselines in test correctness, coverage, and practical utility. Its modular agent design enables adaptive context reconstruction, yielding semantically

Table 9. Function-level CodeBLEU similarity between generated and original tests.

Repository	Overlap Tests	CodeBLEU (%)
node-fs-extra	–	–
node-graceful-fs	2	22.70
node-jsonfile	4	23.04
bluebird	7	17.19
q	–	–
node-dir	1	17.91
pull-stream	6	12.83
plural	–	–
node-dirty	2	15.59
node-geo-point	–	–
node-uneval	1	13.40
Image Downloader	1	20.28
crawler-url-parser	3	20.51
express	1	25.38
yo	6	24.33
eslint	47	29.10
github-readme-stats	3	15.94
jszip	2	20.82
commander.js	7	12.79
mindjs	–	–
Average		**19.45**

grounded and executable tests. Furthermore, function-level similarity analysis confirms that improvements stem from genuine reasoning rather than data memorization, validating the framework's robustness and novelty.

5 Discussions

5.1 Limitations

While code coverage serves as a practical metric for guiding test generation, it remains an imperfect measure of test effectiveness. High coverage does not guarantee strong behavioral assurance, particularly when tests contain trivial assertions. Our evaluation focuses primarily on dynamic execution metrics and does not include static analysis such as test smell detection or mutation testing, which is widely acknowledged as a stronger indicator of fault detection capability. Nevertheless, our approach demonstrates substantial practical value through real bug discovery across multiple projects.

Another limitation concerns LEA's source-code-based library analysis. LEA infers third-party library signatures by reading the JavaScript source code of exported functions. This approach is less effective for packages that expose only thin JavaScript bindings over native C++ extensions, where the binding code itself carries little semantic information. In such cases, LEA degrades gracefully without disrupting the pipeline, but the inferred signatures may be imprecise, potentially reducing test quality for functions that invoke such dependencies. We note that our current benchmark repositories consist predominantly of pure JavaScript libraries, so this limitation had minimal impact on our reported results. A

promising direction for future work is to leverage TypeScript declaration files (`.d.ts`) as a complementary information source, which provide accurate type and signature information independent of the underlying implementation.

5.2 Generalizability Across Languages

We believe that JSTestCraft can generalize across different programming languages, particularly dynamic languages like Python [40], where the deficits of API context, structural context, and semantic context are similarly relevant. Just like JavaScript, Python unit test generation faces difficulties due to dynamic typing, asynchronous patterns, and third-party library integration. While some languages may have mature tools addressing specific aspects of test generation, such as type inference or coverage analysis [46], these tools still struggle with capturing the rich multi-level context required for reliable, executable tests. Our approach, however, is modular and language-agnostic, which means the core framework of context enrichment can be adapted to other languages with minimal modification. This adaptability stems allows the methodology to be transferred to any dynamic language with similar challenges. Thus, although language-specific tools may handle certain tasks more efficiently, JSTestCraft's overall framework remains highly portable and can be readily extended to other programming languages.

5.3 Future Integration with Agent-based Paradigms

Recent progress in agentic LLM architectures provides a promising path for advancing automated test generation. Our framework already exhibits preliminary agent-like behavior through coordinated context enrichment and feedback-driven optimization. In future work, these components could evolve into autonomous agents with persistent memory and adaptive collaboration, enabling continuous learning across iterations. Such an agent-based paradigm would allow test generation systems to dynamically allocate subtasks, such as enrichment, execution, and refinement, while maintaining interpretable and goal-driven coordination. Exploring efficient communication protocols and reward signals for multi-agent cooperation remains an open and valuable direction for future research.

6 Related Work

Traditional unit test generation. Early work by Mirshokraie *et al.* proposed JSEFT [37], which generates JavaScript unit tests by maximizing function coverage and applying mutation-based oracle strategies. Stallenberg *et al.* introduced a type-aware test generation approach that infers variable types probabilistically to better guide test construction [52]. Arteca *et al.* developed *Nessie* [1], which focuses on generating tests for asynchronous JavaScript APIs with nested callbacks. In the Node.js security ecosystem, NODEMEDIC and NODEMEDIC-FINE analyze security vulnerabilities caused by module interactions and asynchronous behaviors [3, 4]. Mesbah *et al.* contributed extensively to automated web and JavaScript testing, including fuzzing, DOM exploration, and client-side failure detection [33]. Beyond these, UnitTestBot JavaScript [32] adopts a fuzzing-based approach with runtime type probing for JavaScript test generation, while SynTest-JavaScript [42] applies search-based techniques using genetic algorithms to explore the input space of JavaScript functions. While these approaches advance automated test generation for JavaScript, they often assume limited semantic variation or do not explicitly reason about cross-function correlations, which is the focus of JSTestCraft.

Program analysis and JavaScript. Beyond pure test generation, a complementary body of work investigates program analysis as a foundation for reasoning about JavaScript behavior. A substantial line of research studies static and dynamic analyses tailored to JavaScript's dynamic features (e.g., aliasing, dynamic property assignment, prototype chains, and asynchronous control flow). Notable static analysis frameworks include TAJS and related type/points-to analyses for JavaScript [18, 19], which provide foundational techniques for whole-program type inference and points-to

reasoning. Tip *et al.* studied scalable typing and semantic approximations for dynamic languages [51], while Kashyap *et al.* refined type information to improve the precision of JavaScript static analysis [24]. These works collectively highlight the importance of deep program analysis, directly motivating JSTestCraft's correlation-analysis component.

LLMs for unit test generation. Recently, LLM-based approaches have gained significant traction in automated software test generation. On the evaluation front, Yang *et al.* and Ouedraogo *et al.* present large-scale empirical analyses of LLMs in test generation, identifying strengths and recurring failure modes [43, 58]. On the method side, approaches such as HITS [54], input-space partitioning [27], and reinforcement-learning refinement [53] illustrate how to systematically improve test coverage and diversity. Despite their promise, these techniques often lack mechanisms for integrating external API knowledge or managing correlation tests, which gaps that JSTestCraft addresses through combined static analysis and context-guided synthesis.

Agent-based test generation. The use of agentic architectures in software testing predates LLMs, with early frameworks employing autonomous agents to coordinate test data generation and execution [59]. Subsequent works extended this idea to simulation-based systems, where multi-agent cooperation was used to explore diverse testing environments [5]. More recently, the resurgence of "agentic AI" has renewed interest in leveraging multi-agent systems for code and test generation. AgentCoder [17] introduced a collaborative framework of coding, testing, and debugging agents for iterative program synthesis, while Code Agents are State-of-the-Art Software Testers [39] empirically demonstrated that coordinated LLM agents can outperform single-model baselines in software testing tasks. Building on this momentum, UnitTenX [6] and AutoMT [28] explore specialized multi-agent LLM frameworks for legacy and metamorphic testing scenarios, respectively, revealing how division of labor and feedback loops among agents enhance test robustness and adaptability. Our work aligns with this new research trend by embedding multiple reasoning agents, each focusing on structural enrichment, semantic understanding, and external API grounding, within a unified generation loop, enabling contextualized and functionally coherent test synthesis.

7 Conclusion

This paper presented JSTestCraft, an agentic, multi-layered framework for automated JavaScript unit test generation that reconstructs missing context via three cooperating enrichment agents. These agents contribute to a persistent Shared Context Memory , which is accessed by the Testing & Optimization Agent to perform retrieval-augmented generation, execution, and iterative optimization. By transforming one-shot test generation into a reasoning–execution loop that continuously enriches external context, structural context, and semantic context, JSTestCraft generates more executable and semantically valid tests. Evaluated across 20 real-world Node.js repositories, JSTestCraft achieves a 60.9% higher test pass rate, 14.2% higher statement coverage, and 47.3% higher branch coverage compared to TestPilot, the strongest baseline. Beyond coverage, JSTestCraft uncovers 13 previously unknown bugs in widely-used projects, 6 of which received substantive acknowledgment from maintainers or community contributors, further highlighting its practical utility.

References

[1] Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. 2022. Nessie: Automatically testing javascript apis with asynchronous callbacks. In *Proceedings of the 44th International Conference on Software Engineering*. 1494–1505.

[2] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[3] Darion Cassel, Nuno Sabino, Min-Chien Hsu, Ruben Martins, and Limin Jia. 2025. NODEMEDIC-FINE: Automatic Detection and Exploit Synthesis for Node. js Vulnerabilities. In *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS'25). doi*, Vol. 10.

[4] Darion Cassel, Wai Tuck Wong, and Limin Jia. 2023. Nodemedic: End-to-end analysis of node. js vulnerabilities with provenance graphs. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 1101–1127.

[5] Greg Chance, Abanoub Ghobrial, Severin Lemaignan, Tony Pipe, and Kerstin Eder. 2020. An agency-directed approach to test generation for simulation-based autonomous vehicle verification. In *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, 31–38.

[6] Yiannis Charalambous, Claudionor N Coelho Jr, Luis Lamb, and Lucas C Cordeiro. 2025. UnitTenX: Generating Tests for Legacy Packages with AI Agents Powered by Formal Verification. *arXiv preprint arXiv:2510.05441* (2025).

[7] Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 201–211.

[8] eslint. 2025. Prototype pollution via internal merging logic. https://github.com/eslint/eslint/issues/19749.

[9] eslint. 2025. Stack overflow in custom rule handling. https://github.com/eslint/eslint/issues/19646.

[10] express. 2025. Charset mismatch due to case sensitivity in content-type. https://github.com/expressjs/express/issues/6613.

[11] express. 2025. Improper handling of extremely long string inputs. https://github.com/expressjs/express/issues/6611.

[12] express. 2025. Unvalidated URLs in res.location lead to open redirects. https://github.com/expressjs/express/issues/6614.

[13] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. 2024. A survey on rag meeting llms: Towards retrieval-augmented large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 6491–6501.

[14] github-readme stats. 2025. Boolean parsing logic misinterprets falsy values. https://github.com/anuraghazra/github-readme-stats/issues/2640.

[15] Lucas Gren and Vard Antinyan. 2017. On the relation between unit testing and code quality. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 52–56.

[16] Siqi Gu, Chunrong Fang, Quanjun Zhang, Fangyuan Tian, and Zhenyu Chen. 2024. Testart: Improving llm-based unit test via co-evolution of automated generation and repair iteration. *arXiv e-prints* (2024), arXiv–2408.

[17] Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010* (2023).

[18] Simon Holm Jensen, Magnus Madsen, and Anders Møller. 2011. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 59–69.

[19] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *International Static Analysis Symposium*. Springer, 238–255.

[20] jszip. 2025. Incorrect encoding of non-ASCII filenames in zip archive. https://github.com/Stuk/jszip/issues/950.

[21] jszip. 2025. Prototype inheritance issue with input validation. https://github.com/Stuk/jszip/issues/951.

[22] jszip. 2025. Shared state persists across zip.clone instances. https://github.com/Stuk/jszip/issues/949.

[23] jszip. 2025. Support for standard Node.js Writable streams. https://github.com/Stuk/jszip/issues/954.

[24] Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. 2013. Type refinement for static analysis of JavaScript. In *Proceedings of the 9th symposium on Dynamic languages*. 17–26.

[25] Raula Gaikovina Kula, Ali Ouni, Daniel M German, and Katsuro Inoue. 2017. On the impact of micro-packages: An empirical study of the npm javascript ecosystem. *arXiv preprint arXiv:1709.04638* (2017).

[26] Ludvig Lemner, Linnea Wahlgren, Gregory Gay, Nasser Mohammadiha, Jingxiong Liu, and Joakim Wennerberg. 2024. Exploring the integration of large language models in industrial test maintenance processes. *arXiv preprint arXiv:2409.06416* (2024).

[27] Jiageng Li, Zhen Dong, Chong Wang, Haozhen You, Cen Zhang, Yang Liu, and Xin Peng. 2024. Llm based input space partitioning testing for library apis. *arXiv preprint arXiv:2501.05456* (2024).

[28] Linfeng Liang, Chenkai Tan, Yao Deng, Yingfeng Cai, TY Chen, and Xi Zheng. 2025. AutoMT: A Multi-Agent LLM Framework for Automated Metamorphic Testing of Autonomous Driving Systems. *arXiv preprint arXiv:2510.19438* (2025).

[29] Runlin Liu, Zhe Zhang, Yunge Hu, Yuhang Lin, Xiang Gao, and Hailong Sun. 2025. LLM-based Unit Test Generation for Dynamically-Typed Programs. *arXiv preprint arXiv:2503.14000* (2025).

[30] Andrea Lops, Fedelucio Narducci, Azzurra Ragone, Michelantonio Trizio, and Claudio Bartolini. 2025. A system for automated unit test generation using large language models and assessment of generated test suites. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 29–36.

[31] Vincent Massol and Ted Husted. 2003. *JUnit in action*. Manning Publications Co.

[32] Aleksei Menshutin. 2025. Path-Minimal Objects in ArkTS Symbolic Execution: From Path Constraints to TypeScript Tests. In *2025 5th International Conference on Code Quality (ICCQ)*. 29–40. doi:10.1109/ICCQ65694.2025.11314570

[33] Ali Mesbah, Arie Van Deursen, and Danny Roest. 2011. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering* 38, 1 (2011), 35–53.

[34] mindjs. 2025. Incorrect promise resolution in event handling logic. https://github.com/mindjs/mindjs/issues/125.

[35] mindjs. 2025. Recursive processing leads to stack overflow (1). https://github.com/mindjs/mindjs/issues/124.

[36] mindjs. 2025. Recursive processing leads to stack overflow (2). https://github.com/mindjs/mindjs/issues/123.

[37] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. Jseft: Automated javascript unit test generation. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.

[38] mochajs. 2011. Mocha. https://mochajs.org/.

[39] Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. 2024. Code agents are state of the art software testers. In *ICML 2024 Workshop on LLMs and Cognition*.

[40] Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gaelli, and Roel Wuyts. 2005. On the revival of dynamic languages. In *International Conference on Software Composition*. Springer, 1–13.

[41] Node.js Contributors. 2024. c8: Modern Code Coverage for Node.js. https://github.com/bcoe/c8.

[42] Mitchell Olsthoorn, Dimitri Stallenberg, and Annibale Panichella. 2024. Syntest-JavaScript: Automated Unit-Level Test Case Generation for JavaScript. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing* (Lisbon, Portugal) *(SBFT '24)*. Association for Computing Machinery, New York, NY, USA, 21–24. doi:10.1145/3643659.3643928

[43] Wendkuuni C Ouedraogo, Kader Kabore, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawende F Bissyande. 2024. Llms and prompting for unit test generation: A large-scale evaluation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 2464–2465.

[44] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84.

[45] Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. 2021. Accelerating JavaScript static analysis via dynamic shortcuts. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1129–1140.

[46] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets deep learning: a hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*. 2019–2030.

[47] Pytest-dev. 2004. Pytest. https://docs.pytest.org/en/stable/.

[48] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[49] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105.

[50] Thodoris Sotiropoulos and Benjamin Livshits. 2019. Static analysis for asynchronous JavaScript programs. *arXiv preprint arXiv:1901.03575* (2019).

[51] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation tracking for points-to analysis of JavaScript. In *European Conference on Object-Oriented Programming*. Springer, 435–458.

[52] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2022. Guess what: Test case generation for Javascript with unsupervised probabilistic type inference. In *International Symposium on Search Based Software Engineering*. Springer, 67–82.

[53] Benjamin Steenhoek, Michele Tufano, Neel Sundaresan, and Alexey Svyatkovskiy. 2025. Reinforcement learning from automatic feedback for high-quality unit test generation. In *2025 IEEE/ACM International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*. IEEE, 37–44.

[54] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1258–1268.

[55] Michael Wooldridge. 2009. *An Introduction to MultiAgent Systems* (2nd ed.). Wiley Publishing.

[56] Michael Wooldridge and Nicholas R. Jennings. 1995. Intelligent agents: theory and practice. *The Knowledge Engineering Review* 10, 2 (1995), 115–152. doi:10.1017/S0269888900008122

[57] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. *arXiv e-prints* (2023), arXiv–2305.

[58] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. 2024. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1607–1619.

[59] Siwen Yu, Jun Ai, and Yifu Zhang. 2009. Software test data generation based on multi-agent. In *International Conference on Advanced Software Engineering and Its Applications*. Springer, 188–195.

[60] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207* (2023).

[61] Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. When language model meets private library. *arXiv preprint arXiv:2210.17236* (2022).

[62] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software unit test coverage and adequacy. *Acm computing surveys (csur)* 29, 4 (1997), 366–427.