Reconstructing Objects for Software Testing

KETAI QIU, Università della Svizzera Italiana (USI), Switzerland
LUCA DI GRAZIA, University of St. Gallen, Switzerland
ALESSIO GAMBI, Austrian Institute of Technology, Austria; the Italian Institute of Artificial Intelligence (AI4I), Italy
MATTIA FAZZINI, University of Minnesota, USA
MAURO PEZZÈ, Università della Svizzera Italiana (USI), Switzerland and University of Milano-Bicocca, Italy

Software testing requires developers to build nontrivial object states to thoroughly exercise programs that process data structures. Manually building these test cases is an effort-demanding activity. The execution of software in production is an enormous source of program states, which can be serialized to synthesize test inputs. Unfortunately, most serialization approaches prioritize performance over transparency, making it difficult to generate useful and maintainable test cases. We present X2J, an approach that captures objects from runtime and automatically generates human-readable code that reconstructs those objects. X2J leverages Large Language Models, enhanced with an AST-aware Retrieval-Augmented Generation component, and an iterative Validation-Driven Repair pipeline that seamlessly aligns the generated code with the captured state. We evaluated X2J across two benchmarks that include 10,794 objects from 21 Java projects, achieving 80.7% reconstruction accuracy, that significantly outperforms ProDJ, the closest approach based on program analysis, and six LLM baselines. We further validated X2J on a closed-source industrial application, and X2J achieved 89.5% accuracy, which confirms the results obtained using open-source software and supports the practical applicability of the approach. Overall, our evaluation shows that X2J successfully transforms production objects into maintainable test cases, thereby enhancing the efficiency and effectiveness of software testing.

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

Additional Key Words and Phrases: Runtime Object Reconstruction, Large Language Model, Field Testing

ACM Reference Format:
Ketai Qiu, Luca Di Grazia, Alessio Gambi, Mattia Fazzini, and Mauro Pezzè. 2026. Reconstructing Objects for Software Testing. 1, 1 (November 2026), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The quality of software testing depends on both the variety of the test inputs [63] and the precision of the assertions that check the resulting program states [39], while the usefulness of the test suites depends on the readability and maintainability of the test cases [14]. The execution of software in production offers rich traces of runtime objects, which can be used to generate useful and thorough test suites [18, 19, 45]. Manually crafting tests that exercise complex objects observed in production is notoriously expensive [35, 50]. Although automated test generation has

Authors' Contact Information: Ketai Qiu, Università della Svizzera Italiana (USI), Lugano, Switzerland, ketai.qiu@usi.ch; Luca Di Grazia, University of St. Gallen, St. Gallen, Switzerland, work@lucadigrazia.com; Alessio Gambi, Austrian Institute of Technology, Austria; the Italian Institute of Artificial Intelligence (AI4I), Turin, Italy, alessio.gambi@ait.ac.at; Mattia Fazzini, University of Minnesota, Minneapolis, USA, mfazzini@umn.edu; Mauro Pezzè, Università della Svizzera Italiana (USI), Lugano, Switzerland and University of Milano-Bicocca, Milan, Italy, mauro.pezze@usi.ch.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

advanced substantially [20, 56, 59], search-based and pseudo-random approaches struggle to reconstruct realistic object states [34]. In this paper, we propose an approach to automatically reconstruct precise object states from production traces to generate useful test cases. When executed in production, modern software applications, like microservices, web applications, and utility libraries, instantiate a large number of objects that are typically encapsulated as in-memory objects [15], and that can be used to enhance the test suites [45]. Moreover, as software evolves continuously through rapid releases, feature toggles, and data-driven adaptations, important behaviors often emerge only *after* deployment [8, 23]. Collecting and reusing data from program states observed in production can largely improve the quality of the test suites and properly exercise states that emerge only in production.

Automatically transforming complex data from production-observed program states into maintainable test suites remains a largely open problem. State-of-the-art approaches are able to instantiate common object graphs with various fields. However, these approaches still present several shortcomings: (i) binary serialization yields unreadable and non-refactorable test code [7]; (ii) rule-based approaches like CoDeSe [24] rely on rigid getter/setter templates that frequently fail to infer correct generic types or nested collection instantiation; (iii) instrumentation-based tools, such as Darbord et al.'s approach [12] and ProDJ [58], require intrusive instrumentation or semi-automated trace extensions that are impractical in production. The search space for reconstructing complex objects can also be vast due to their structural depth and nested dependencies. Consequently, developers often revert to manual object creation, losing the fidelity of production states.

We propose X2J, an approach that reconstructs serialized runtime objects into human-readable test suites for Java applications. X2J encodes the objects observed in production states in semi-structured formats such as XML, retrieves the source code that is relevant for building the observed objects, and automatically synthesizes executable Java code that reconstructs the original objects. X2J targets the open research challenge of reconstructing objects serialized at runtime by balancing *validity*, *semantic equivalence*, *readability*, and *transparency*, as detailed in Section 2. X2J addresses the limitation of binary serialization and heavy tracing mechanism by leveraging the reasoning capabilities of Large Language Models (LLMs) [1, 41, 60].

We evaluated X2J across two benchmarks for a total of 10,794 unique objects, and obtained 80.7% average reconstruction accuracy. X2J achieves 75.1% accuracy on JObjectsBench, an open-source dataset with 5,046 objects collected from 17 open-source Java projects, and 85.4% accuracy on the 5,748 objects in the benchmark that X2J shares with ProDJ, the closest related approach, which obtains 70.9% accuracy. We also evaluated X2J on a closed-source industrial web application. X2J achieves 89.5% accuracy, thus confirming the results obtained on the open-source benchmarks and demonstrating its applicability in an industrial context. We evaluated the usefulness of X2J to reconstruct object states that reveal bugs. On 51 bugs that we sampled from Defects4J, X2J successfully reproduced 17 failures, five times more than those reproduced by Libro [29]. The inspection of the reproduced failures indicates that X2J and Libro are largely complementary, as they rely on different sources of information. In particular, X2J can reconstruct complex object states that are difficult to infer from bug reports alone. Overall, our experiments indicate that X2J substantially reduces the developer's effort in creating effective test cases with serialized runtime data, and bridges the gap between legacy hand-crafting testing workflows and automated human-readable code generation, thereby enhancing the quality of test suites.

This paper makes the following contributions:

- X2J, a novel approach that reconstructs object states from production executions into human-readable test suites for Java applications, by integrating AST-Aware RAG with an iterative Validation-Driven Repair pipeline. We release X2J as open-source software with a complete replication package [44].
- JObjectsBench, a benchmark comprising 5,046 unique objects instantiated from the execution of 829 classes of 17 open-source Java projects.
- A comprehensive empirical validation across two benchmarks and a closed-source industrial application, demonstrating that X2J consistently achieves high reconstruction accuracy across varying object complexities and effectively captures runtime states for automated test generation.

The remainder of the paper is structured as follows: Section 2 introduces the terminology and an illustrative example; Section 3 details the X2J approach: Object Capture, AST-Aware RAG, LLM Inference, and Validation-Driven Repair; Section 4 presents the empirical evaluation, including dataset, baselines, results, costs, and threats to validity; Section 5 discusses related work; Section 6 concludes and outlines future directions.

2 Terminology & Illustrative Example

To ground our work, we introduce key terms and present an example that characterizes the task of generating executable code to reconstruct objects created during software execution.

We refer to the structure that represents the relationship between encapsulated objects as an *object graph* [33]. Given an object graph $\mathcal{G}$, an object instance in memory in our case, and its serialized representation $\mathcal{S}$, we define the reconstruction task as a mapping: $\mathcal{R} : \mathcal{S} \longmapsto \mathcal{P}$ where $\mathcal{P}$ denotes the generated Java program that reconstructs an object graph $\mathcal{G}'$ equivalent to $\mathcal{G}$. We analyze the quality of the code generated for an object reconstruction task through the following criteria:

(1) *Validity*: $\mathcal{P}$ compiles successfully and terminates normally.
(2) *Semantic Equivalence*: $\mathcal{G}'$ is semantically equivalent to $\mathcal{G}$, that is, it preserves every field and property.
(3) *Readability*: $\mathcal{P}$ adheres to idiomatic Java conventions, that is, meaningful identifiers, minimal boilerplate, and structured control flow.
(4) *Transparency*: $\mathcal{R}$ relies only on the program source code (type information, method signatures, and package declarations) without intrusive operations like runtime tracing, reflection and heavy instrumentation.

Listing 1 reports the serialization of a `Broadleaf`[1] object ($\mathcal{G}$) that X2J captured in XML format with `XStream` [61], a serialization framework widely used in research and industry [6, 17, 22]. The object $\mathcal{G}$ of type `FulfillmentGroupItemImpl` contains a nested field `orderItem` of type `DiscreteOrderItemImpl`, which contains a `SkuImpl` object, and a set of objects of standard Java types (`String`, `List`, `BigDecimal`). `XStream` maps unassigned fields to empty elements containing only reference metadata (such as the `id` attribute) while omitting the field values (Lines 10-16). Reconstructing the `FulfillmentGroupItemImpl` object state requires both instantiating the correct types and capturing nested object fields that govern the object runtime behavior and dependency resolution.

Listing 2 and Listing 3 report the code ($\mathcal{P}$) of the `FulfillmentGroupItemImpl` object as reconstructed with X2J and ProDJ, respectively. X2J (Listing 2) correctly sequences instantiation and configuration of the complete object graph, by instantiating `DiscreteOrderItemImpl` and `SkuImpl` with precise values, and wiring them using standard setters before assigning the fully initialized `orderItem` to the `FulfillmentGroupItemImpl` object. X2J assembles these

[1] an open-source e-commerce web application in the ProDJ benchmark.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<org.broadleafcommerce.core.order.domain
    .FulfillmentGroupItemImpl id="1">
    <orderItem class="org.broadleafcommerce.core
        .order.domain.DiscreteOrderItemImpl" id="2">
        <auditable id="3"/>
        <quantity>2</quantity>
        <retailPrice>1.00</retailPrice>
        <salePrice>1.00</salePrice>
        <orderItemAdjustments id="4"/>
        <proratedOrderItemAdjustments id="5"/>
        <orderItemQualifiers id="6"/>
        <candidateItemOffers id="7"/>
        <orderItemPriceDetails id="8"/>
        <orderItemAttributeMap id="9"/>
        <childOrderItems id="10"/>
        <baseRetailPrice>1.00</baseRetailPrice>
        <sku class="org.broadleafcommerce.core
            .catalog.domain.SkuImpl" id="11">
            <retailPrice>1.00</retailPrice>
            <dimension id="12"/>
            <weight id="13"/>
            <skuMedia id="14"/>
            <legacySkuMedia id="15"/>
            <skuAttributes id="16"/>
            <productOptionValueXrefs id="17"/>
            <legacyProductOptionValues id="18"/>
            <fees id="19"/>
            <fulfillmentFlatRates id="20"/>
            <excludedFulfillmentOptions id="21"/>
            <quantityAvailable>0</quantityAvailable>
        </sku>
        <additionalAttributes id="22"/>
        <discreteOrderItemFeePrices id="23"/>
    </orderItem>
    <quantity>2</quantity>
    <taxes id="24"/>
</org.broadleafcommerce.core.order.domain
    .FulfillmentGroupItemImpl>
```

Listing 1. Serialized object in XML format

```java
@Test
public void testLLMGeneratedCode() throws Exception {
    FulfillmentGroupItemImpl target_object =
        new FulfillmentGroupItemImpl();
    target_object.setQuantity(2);
    OrderItem orderItem = new DiscreteOrderItemImpl();
    orderItem.setQuantity(2);
    orderItem.setRetailPrice(
        new Money(BigDecimal.valueOf(1.00)));
    orderItem.setSalePrice(
        new Money(BigDecimal.valueOf(1.00)));
    Sku sku = new SkuImpl();
    sku.setRetailPrice(
        new Money(BigDecimal.valueOf(1.00)));
    ((DiscreteOrderItemImpl) orderItem).setSku(sku);
    target_object.setOrderItem(orderItem);
}
```

Listing 2. Object reconstruction code generated with X2J

```java
@Test
void test49HashCode$() {
    // Arrange
    FulfillmentGroupItemImpl receiver =
        new FulfillmentGroupItemImpl();
    // Act
    int actual = receiver.hashCode();
    // Assert
    int expected = 872443152;
    AssertJEqualityHelper
        .collectStatistics(actual, expected);
    assertThat(actual).isEqualTo(expected);
}
```

Listing 3. Object reconstruction code generated with ProDJ

dependencies into a properly typed object hierarchy, configures nested field contracts, and injects the fully initialized dependencies via type-safe casting and setter calls. This sequence of method calls preserves both structural topology and execution-time constraints. Crucially, the generated code is structured as a standalone JUnit [53] test that can be directly incorporated into existing test suites without refactoring.

ProDJ (Listing 3) generates a minimal receiver instantiation without reconstructing the dependencies or configuring the nested fields. ProDJ successfully infers the correct target type; however, its reliance on action-based templates fundamentally limits it to shallow object verification. Consequently, the generated setups are structurally complete but functionally inert: they pass compilation and verify isolated method contracts but fail to wire the complex object graphs or reproduce the contextual states required for testing. This limitation underscores a challenge in template-based reconstruction: modern applications rely on interconnected domain models where correct dependency provisioning and field values are essential for reproducing runtime behaviors.

The example shows how X2J reconstructs a nested object graph and generates a test case that can be directly integrated into an existing test suite. Unlike conventional reconstruction techniques that often instantiate objects without fully restoring their nested dependencies and framework-specific configurations, X2J reproduces the runtime state observed

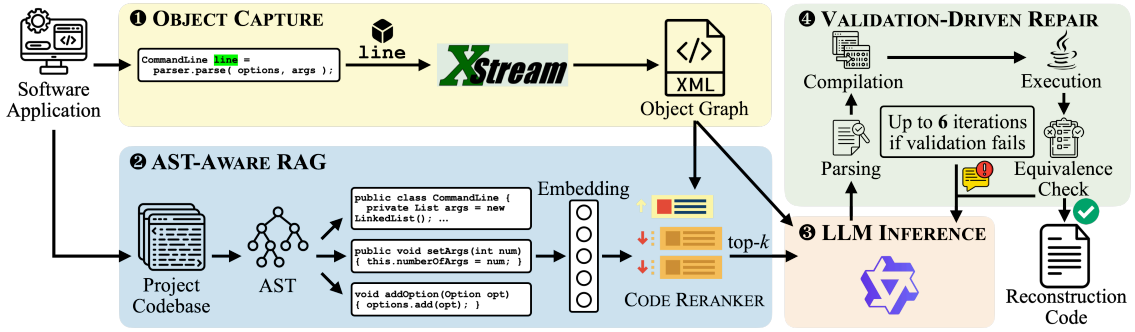


Fig. 1. Overview of X2J's LLM-based pipeline to reconstruct Java objects from their serialized form

in production executions and encodes it as executable test setup code. As a result, developers can reuse application states observed in production without manually recreating complex initialization logic or dependency relationships. By bridging production execution traces and automated test scaffolding, X2J enables test cases to exercise execution conditions that are difficult to reproduce manually and are frequently omitted by conventional object reconstruction approaches. The next section describes the design of X2J and its object reconstruction process.

3 Approach

We present X2J, an LLM-based approach for generating human-readable Java code that reconstructs the state of runtime objects. The Java code can be used as part of the prefix unit tests. X2J automatically generates Java code from a serialized object and the source code of the target project. The generated code instantiates the object in the original state. Figure 1 shows the four steps of X2J: (i) **Object Capture** serializes the in-memory object into XML format; (ii) **AST-Aware RAG** retrieves the relevant context code and associated Abstract Syntax Tree (AST) from the project; (iii) **LLM Inference** prompts a large language model with the captured object and the retrieved context to synthesize Java code that recreates the original object state; finally, (iv) **Validation-Driven Repair** parses, compiles, executes and iteratively repairs the generated code until it satisfies the equivalence check or the computational budget is exhausted. The following subsections describe each step in detail.

3.1 Object Capture

Object Capture transforms the in-memory Java object into a form that represents the original state of the object. Specifically, it serializes the runtime object into XML format [4, 57], a format that Beyzaei et al. suggest for readability in software testing research [9], using XStream [61] library. We selected XStream as the serializer for capturing object states because XML is a widely used serialization format in industrial Java contexts [6], and XStream preserves references, generics, and primitives in a format that LLMs handle reliably. The downstream AST-Aware RAG and Validation-Driven Repair components are format-agnostic; thus, porting X2J to other serializers would only require reimplementing Object Capture and the equivalence-check component.

Listing 1 shows the serialized object that Object Capture produces from Broadleaf. The XML code of Listing 1 captures the hierarchical structure of the object and the leaf component of the object, which is enough for X2J to generate the Java object of Listing 2 that, once executed, yields an object semantically equivalent to the original object.

3.2 AST-Aware RAG

LLMs often generate *hallucinated* code that calls nonexistent functions or uses undeclared symbols, limiting its effectiveness in complex projects [68]. X2J addresses the hallucination problem by pairing LLM with an Abstract Syntax Tree aware Retrieval Augmented Generation (AST-Aware RAG) component, which retrieves constructors, methods, and fields relevant to the target object, and an iterative Validation-Driven Repair pipeline, which parses, compiles, executes, and semantically compares the reconstructed object to the original one, feeding back errors and differences to the LLM until convergence or the computational budget is exhausted.

Most prior work leveraged Abstract Syntax Trees (ASTs) to contextualize LLMs for test generation [11, 25, 46, 49], and only few approaches addressed the specific challenge of reconstructing complex object graphs into production-grade test suites. AST-Aware RAG is not merely a code preprocessing parser but a pivotal component that substantially mitigates LLM hallucinations during code generation for object reconstruction by preserving the semantics of code fragments. The core contribution of this paper is the *synergistic combination of the iterative Validation-Driven Repair loop with AST-Aware RAG*, which bridges a fundamental semantic gap: translating serialization formats into structure-aware Java test suites that reconstruct the object states of production executions.

AST-Aware RAG extracts the syntactic structure of Java source code (excluding the tests) by parsing the `.java` source files into Abstract Syntax Trees (AST) and segmenting them into code-aware segments. Each segment contains complete class definitions, individual method bodies, or field declarations annotated with the originating package and class names (up to 1024 tokens). AST-Aware RAG is invoked for every LLM prompt to dramatically suppress hallucinations in the generated code.

AST-Aware RAG infers the relevance between the input XML document and each code segment by combining the normalized cosine similarity (ranging from 0 to 1) of token embeddings with a heuristic Code Reranker. The Code Reranker prioritizes the code segments that are likely to be required for reconstructing the object, such as method and field declarations. It assigns a weight p to the code segments whose package matches the serialized object's package, a weight c to the code segments whose package matches a fully qualified class name, a weight m to the code segments containing method declarations, and a weight f to the code segments containing field declarations. The Code Reranker computes the score of the segments by multiplying the default weights of the segments with the cosine similarities: For a segment s with similarity $sim(s)$ and weights $W(s) = \{p, c, m, f\}$, the reranked score is $score(s) = sim(s) \prod_{w \in W(s)} w$. For example, AST-Aware RAG assigns the two weights $c \times f$ to a code segment that both matches the class and contains a field declaration. The weights are greater than or equal to 1, so the Code Reranker can only promote segments relative to their original similarity score. Setting $p = c = m = f = 1$ exactly recovers the original unweighted ordering. These heuristics are grounded in the observation that an object's state is most frequently established within factory methods, constructors, and field initializers. A preliminary experiment on random 229 objects confirms the improvement of reconstruction accuracy (+7.9%) over the configuration without Code Reranker ($p = 1, c = 1, m = 1, f = 1$). We set $p = 1.2, c = 1.6, m = 1.2, f = 1.2$ by default and make them configurable by users.

AST-Aware RAG selects the top-k segments after ranking, where k is determined with respect to the LLM's context window. In our evaluations, we set a maximum limit of 4K tokens for each code retrieval process to ensure a fair comparison across different LLMs. AST-Aware RAG is invoked before making any request to the LLM, i.e., before the initial request to reconstruct the serialized object and before any retry following a failed validation. In the latter case, AST-Aware RAG performs an error-guided retrieval. For instance, if a generation attempt fails due to an undefined symbol, AST-Aware RAG uses the corresponding error message to retrieve the segments mentioning the missing

symbol. AST-Aware RAG aims at ensuring that the prompt always contains the most relevant portions of the codebase, thus substantially reducing hallucinations and improving the generated reconstruction code's quality.

The motivation for using a AST-Aware RAG comes from the limitations of LLMs and vanilla retrieval-augmented generation (RAG). LLMs are prone to hallucinations [30], often producing code that is syntactically valid yet semantically incorrect. While RAG mitigates this by supplementing prompts with retrieved context [67], classic text-based RAG treats source code as flat token sequences and ignores its structural relationships [65]. As we show in our evaluation, for Java, this omission can cause retrieval to miss essential package, class, or method context needed for reconstructing serialized objects. Although modern LLMs support context lengths up to millions of tokens [13, 37, 52], prompting with entire codebases is prohibitively slow and risks distracting the model, degrading accuracy [67]. AST-Aware RAG directly addresses these challenges by retrieving only the most relevant code snippets instead of overwhelming the LLM with unfiltered project-wide context.

3.3 LLM Inference

LLM Inference builds the prompts to reconstruct the serialized object. We follow best practices in prompt engineering for code generation [47] and design a prompt template with clear instructions for the LLM as shown in Figure 2[2].

The **System Message** section defines the LLM's role as a Java expert and the task to reconstruct the object state. The section enumerates a set of requirements for the code generation task. These constraints ensure that the generated code (i) instantiates the object described by the XML such that they have identical field values, (ii) uses only classes and methods available in the provided context, (iii) encapsulates all the code inside a method with a pre-defined signature, and (iv) excludes LLM-generated comments, package declarations and class declarations. The **User Message** section contains two parts: the XML document that represents the object state, and the code context AST-Aware RAG retrieved. In case the validation step fails (see Section 3.4), X2J appends to the prompt a **Repair Message**. The Repair Message contains the context code retrieved by AST-Aware RAG and a description of the issue to fix. For each object reconstruction task, X2J maintains a continuous conversation with the LLM and retains all previous messages, including the code contexts retrieved by AST-Aware RAG and the repair messages from previous iterations. Thus, each iteration extends the existing conversation with the context and validation error relevant to the current iteration. Figure 2 exemplifies the case of a compilation error caused by an unresolved symbol.

3.4 Validation-Driven Repair

Generating correct Java code is challenging, and LLM Inference may occasionally fail to generate correct Java code. Validation-Driven Repair is X2J's mechanism to recover from these situations and guarantee both syntactic and semantic correctness of the generated Java code.

Validation-Driven Repair consists of four steps: parsing, compilation, execution, and equivalence checking. If all four validation steps pass, X2J returns the generated code as correct. However, when the validation steps identify a problem, X2J enters an iterative loop to fix the issues. Inspired by recent research results [10], X2J collects validation errors up to 1024 tokens, generates a "repair message" based on them, and passes it to the LLM as an additional context before attempting to regenerate the code. This loop terminates if the updated code is correct, the generation budget (e.g., six iterations) is exhausted, or there is no progress (i.e., the same validation error appears more than three times).

[2]The detailed prompt is available in the X2J replication package.

System Message

```
# Role
You are an expert in Java 8 programming and object serialization.
# Task
Your task is to generate Java 8 code that instantiates an object identical to the one serialized in the provided XML.
You must analyze the XML elements (generated by the `XStream` library) and write code that recreates this exact object state.
# Inputs
You will be provided with two elements:
* **XML**: The serialized representation of the Java object.
* **CONTEXT**: The relevant Java classes and methods available for you to use.
# Constraints
* **Restricted Scope**: You must use **only** the classes and methods explicitly declared in the `CONTEXT` (along with standard Java 8 libraries).
* **Exact Match**: The generated object must have the exact same attributes and values as the one defined in the `XML`.
* **Template Adherence**: You must strictly use the following Java code template.
Add your complete Java import statements at the top, and replace `// PUT CODE HERE` with your object reconstruction logic:
```Java
public class ObjectReconstructionTests {
    @Test
    public void testLLMGeneratedCode() throws Exception {
        // PUT CODE HERE
    }
}
```
```

User Message

XML

```
<org.broadleafcommerce.cms
    .page.service.PageSiteMapGenerator>
  <pageDao class="dynamic-proxy" id="2">
    <interface>
      org.broadleafcommerce.cms.page.dao.PageDao
    </interface> ...
    <className>
      org.broadleafcommerce.cms.page.domain.Page
    </className>
    <methodName>getFullUrl</methodName>
    <val_-value class="string">
      /about_us</val_-value> ...
    <className>
      org.broadleafcommerce.cms.page.domain.Page
    </className>
    <methodName>getFullUrl</methodName>
    <val_-value class="string">faq</val_-value> ...
    <className>
      org.broadleafcommerce.cms.page.domain.Page
    </className>
    <methodName>getFullUrl</methodName>
    <val_-value class="string">
      /new-to-hot-sauce</val_-value>
  </pageDao>
  <rowLimit>0</rowLimit>
</org.broadleafcommerce.cms
    .page.service.PageSiteMapGenerator>
```

Context

```
@Component("blPageSiteMapGenerator")
public class PageSiteMapGenerator
  implements SiteMapGenerator {
  @Resource(name = "blPageDao")
  protected PageDao pageDao;
  @Value("${page.site.map.generator.row.limit}")
  protected int rowLimit;
  @Override
  public boolean canHandleSiteMapConfiguration(
    SiteMapGeneratorConfiguration
      siteMapGeneratorConfiguration) {
      return SiteMapGeneratorType.PAGE.equals(
        siteMapGeneratorConfiguration
          .getSiteMapGeneratorType());
  }
  @Override
  public void addSiteMapEntries(
    SiteMapGeneratorConfiguration smgc,
    SiteMapBuilder siteMapBuilder) {
      int rowOffset = 0;
      List<Page> pages;
      String previousUrl = "";
      do {
        pages = pageDao.readOnlineAndIncludedPages(
          rowLimit, rowOffset, "fullUrl");
        rowOffset += pages.size();
...// Omitted for brevity
```

Prompt

```
# Task
Write Java 8 code that creates the exact same object
  ({object_name}) as serialized in the provided XML.
# Inputs
- XML: The serialized object data.
- CONTEXT: The specific environment of allowed classes
  and methods.
- Target Variable Name: {object_name}
- Target Class Type: {class_name}
# Constraints
- Restricted Environment: You must ONLY use classes and
  methods explicitly declared in the provided CONTEXT.
- Validation Requirement: Before outputting the code,
  double-check that every single method you call is
  declared in the CONTEXT.
- Imports: You must generate complete import statements
  for all used names at the top of the file. Infer these
  from the CONTEXT and standard Java 8 libraries.
- General Knowledge: You may use standard Java 8
      programming
  knowledge for standard language features, but custom
  objects/methods must strictly come from the CONTEXT.
- No Comments: Do NOT include any comments in the
      generated code.
# Output Format
Provide ONLY the Java code using the given template.
  Do not include any explanations outside the code block.
```

Repair Message

Context

```
public interface PageDao {
  Page readPageById(Long id);
  List<PageField> readPageFieldsByPageId(Long pageId);
  PageTemplate readPageTemplateById(Long id);
  PageTemplate savePageTemplate(PageTemplate template);
  Page updatePage(Page page);
  void delete(Page page);
  Page addPage(Page clonedPage);
  List<Page> readAllPages();
...// Omitted for brevity
```

Prompt

```
Fix following errors with CONTEXT. You can ONLY use methods defined in CONTEXT.
[ERROR] /Projects/BroadleafCommerce/admin/broadleaf-contentmanagement-module/src/test/java/
       org/broadleafcommerce/cms/page/service/ObjectReconstructionTests.java:[30,22]
       cannot find symbol
  symbol:  method setId(long)
  location: variable target_object of type org.broadleafcommerce.cms.page.service.
         PageServiceUtility
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.8.1:
       testCompile (default-testCompile) on project broadleaf-contentmanagement-module:
       Compilation failure
```

Fig. 2. The Prompt template instantiated with the example to reconstruct the `PageSiteMapGenerator` object

Parsing. The parsing step ensures that the code is syntactically valid and fully encapsulated within the required output method (`testLLMGeneratedCode`). We use the `javalang` parser and a regular-expression filter to detect incorrect

```
<Message> ::= <ChangeType> <ChangeValue> <AddDictItems> <RemoveDictItems> <AddIterableItems>
    <HandleRemainingKeys>
<ChangeType> ::= [ "change the type of" <Field> "to" <Type> { "change the type of" <Field> "to"
    <Type> } ]
<ChangeValue> ::= [ "change the value of" <Field> "to" <Value> { "change the value of" <Field>
    "to" <Value> } ]
<AddDictItems> ::= [ "add the following fields or values:" <DictItems> ]
<RemoveDictItems> ::= [ "set the following fields or values to null:" <DictItems> ]
<DictItems> ::= <DictItem> { <DictItem> }
<DictItem> ::= "object" <Element> { "in" <Element> }
<AddIterableItems> ::= [ "add the following fields or values:" <IterableValues> ]
<HandleRemainingKeys> ::= { <Key> ":" <Value> }
<Field> ::= <Identifier>  <Value> ::= <Literal>      <IterableValues> ::= <ListRepresentation>
<Type> ::= <Identifier>   <Element> ::= <Identifier>  <Key> ::= <Identifier>
```

Listing 4. Diff-to-Action Backus–Naur Form (BNF) grammar.

method signatures and extraneous code outside the method. If the validation fails (e.g., because `javalang` returns a `ParseException` $\mathcal{E}$), X2J generates the following repair message: *"Fix Java syntax errors $\mathcal{E}$. Ensure the method signature containing the generated code is public void testLLMGeneratedCode() throws Exception"*.

Compilation. The compilation step ensures that the code can be compiled in the context of the target project. X2J synthesizes a lightweight test class using `JavaParser`[3] and compiles it with the standard `javac` compiler. If compilation fails, X2J includes in the repair message only the error messages extracted from the compiler logs. Each error message describes the location of the faulty code and the exact error, such as *cannot find symbol* error in Figure 2.

Execution. The execution step checks that the method generated by X2J executes without triggering exceptions and creates a non-null object. In case of failures, X2J collects the complete stack trace and the execution log, filters out the stack trace entries that belong to external libraries, and generates the repair message: *"Fix the following execution errors $\mathcal{E}$"*, where $\mathcal{E}$ contains the error log and the filtered stack trace. The execution of `testLLMGeneratedCode` also produces a serialized version of the reconstructed object graph $\mathcal{G}'$, thus enabling object graph equivalence checking.

Equivalence Checking. The equivalence checking step validates that $\mathcal{G}'$ is equivalent to $\mathcal{G}$ (i.e., the original object graph) by comparing the corresponding XML documents. Specifically, we use xmldiff [4] to compute the *action distance* (counting the minimal number of edit actions) and DeepDiff [5] to summarize the differences between XMLs in a form amenable for automatic processing (i.e., a diff dictionary of mismatched tags and values). This combination of diff tools allows us to check both structural and value equivalence of object graphs in their serialized XML format. Action distance is an edit distance metric that contains the minimal number of edit actions (e.g., XML node insert, XML node delete) required to transform one XML document into the other in a standard format. Consequently, although action distance provides valuable feedback to the LLM, it cannot be directly used as a repair message. Thus, we start from the DeepDiff output and we generate a concise set of natural-language repair directives using a rule-based algorithm based on a Diff-to-Action grammar as shown in Listing 4. For instance, when reconstructing an `orderItem` object in the `Broadleaf` project, the diff dictionary indicates that the original object contains a `quantity` field of value 2, whereas the generated code defaults it to 0. In this case, X2J prompts the LLM with the following repair message: *"change the value of root['org.broadleafcommerce.core.order.domain.FulfillmentGroupItemImpl']['orderItem']['quantity'] to 2"*.

[3] https://github.com/javaparser/javaparser
[4] https://github.com/Shoobx/xmldiff
[5] https://github.com/seperman/deepdiff

4 Empirical Evaluation

We assess the effectiveness, efficiency, and usefulness of X2J, by investigating the following research questions (RQs):

- **RQ1: Does X2J effectively reconstruct objects?** We evaluate the ability of X2J to reconstruct Java objects by counting the number of objects that X2J exactly reconstructs.
- **RQ2: What is the contribution of each component to the effectiveness of X2J?** We conduct an ablation study to evaluate the contribution of each component in X2J by either removing the component or replacing it with a naïve implementation.
- **RQ3: Is X2J cost-effective?** We assess the costs of X2J by computing the number of tokens provided to the LLM, the time spent executing each component of X2J, and the number of iterations of the Validation-Driven Repair loop.
- **RQ4: Is X2J applicable in practice?** We evaluate the practical usefulness of X2J in terms of both code quality and ability to generate tests that reproduce industrially-relevant bugs. We assess code quality by measuring the cognitive complexity, and evaluate the ability to reproduce bugs by counting the number of bugs for which X2J generates a test that successfully reproduces the bug. We evaluate the ability of X2J to reproduce objects observed when executing closed-source code by reporting the results of experimenting with a proprietary application.

4.1 Methodology

Datasets. We evaluate X2J on 10,794 unique objects that we obtained from 21 Java projects through two benchmarks: (i) JObjectsBench, a dataset of 5,046 objects that we built from 17 open-source Java projects; (ii) the ProDJ benchmark [58], contributing 5,748 objects from four projects by replaying the same application-level workloads as the original evaluation. In addition, we validate X2J on `ANON`, a closed-source industrial web application, using 840 captured objects to assess the applicability of X2J in an industrial setting.

The JObjectsBench Dataset. We created JObjectsBench, a dataset of 5,046 *unique* Java objects serialized in XML. We collected the objects from the 17 open-source Java projects of Defects4J v2.0.1 [28], a benchmark widely used in software engineering research. We excluded trivial objects instantiated from Java primitive wrapper classes in the dataset, such as Integer, Byte and Double.

X2J can reproduce any serialized objects; however, since Defects4J projects are libraries, we leverage existing JUnit test cases to collect objects that expose different behaviors. Additionally, sampling runtime objects from test executions is also beneficial to our evaluation for two reasons. On the one hand, it allows us to evaluate X2J under different execution paths. On the other hand, unit tests provide compact code for creating and setting the objects, which allows us to compare the code generated with X2J against human-written code.

We obtain JObjectsBench by running an automatic pipeline to (i) identify all the expressions that either create or modify objects in JUnit tests, (ii) inject code statements to serialize objects to XML files using XStream, and (iii) execute the modified JUnit tests to serialize the runtime objects. We discard objects that exceed 32K LLM tokens in their XML format to fit the context window of all experimented LLMs. The discarded objects account for less than 1% of all captured objects.

Table 1 summarizes the statistics of JObjectsBench. The dataset contains a total of 5,046 serialized objects (col. *#Objects*) from 829 classes (col. *#Classes*) across 17 Java projects (col. *Project*). The size of XML serialized objects ranges between 35 and 139,989 characters (col. *#Characters in XML*). We measure the complexity of captured objects in terms of nesting levels (col. *Object Depth*) and the number of fields in the corresponding class (col. *#Fields in Class*). Intuitively, setting up objects with several non-null and non-primitive attributes requires more effort than setting up objects with

Table 1. Statistics of JObjectsBench and sampled objects from a closed-source, industrial web app (ANON).

Project	#Objects	#Classes	#Characters in XML			Object Depth			#Fields in Class		
			Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Chart	1,021	275	35	134,897	5,551	1	22	6	1	14	4
Cli	153	7	108	4,325	905	2	7	4	1	11	4
Closure	111	18	108	139,989	18,042	1	22	7	1	15	4
Codec	67	7	46	23,595	939	1	2	1	1	2	1
Collections	311	49	139	2,733	536	1	10	4	1	5	2
Compress	107	17	142	32,359	3,457	1	9	4	1	17	7
Csv	66	5	325	9,487	3,350	2	5	3	3	11	5
Gson	222	34	75	3,572	478	1	10	4	1	17	3
JacksonCore	240	24	111	25,633	5,437	1	8	5	1	29	11
JacksonDatabind	220	53	35	29,226	1,905	1	18	5	1	12	4
JacksonXml	16	3	550	8,785	5,603	3	7	6	4	12	9
Jsoup	133	6	98	7,482	4,176	2	20	13	1	8	3
JxPath	171	9	120	28,425	6,678	2	12	6	1	7	2
Lang	289	37	119	12,011	966	1	35	4	1	20	4
Math	898	209	117	105,928	5,478	1	21	3	1	33	4
Mockito	102	31	95	87,870	9,270	1	21	7	1	9	2
Time	919	46	85	8,844	1,103	1	24	6	1	8	2
Total	**5,046**	**829**	**35**	**139,989**	**3,845**	**1**	**35**	**5**	**1**	**33**	**4**
ANON (discussed in RQ4)	840	62	112	4,155	240	1	8	2	1	13	4

few or primitive attributes. Both metrics confirm that JObjectsBench contains non-trivial objects with five nesting levels and four fields on average. Thus, JObjectsBench provides a strong basis to assess the capability of X2J to reconstruct objects.

The ProDJ Benchmark. We evaluate X2J on all four Java projects used in the ProDJ evaluation [58]. We replay the same execution workloads as the original evaluation and capture 5,748 objects. The states of these objects originate from the replayed application-level workloads rather than from the test suites of the projects. Therefore, these objects allow us to assess X2J on program states without corresponding tests. Replaying the original workloads also enables a direct comparison between X2J and ProDJ under identical execution conditions, with ProDJ configured as originally designed.

The ANON application. We evaluate the industrial applicability of X2J on an industrial Java web application (ANON). The source code of ANON is proprietary and thus not publicly available or included in existing open-source datasets, reducing the risk of data leakage from the LLM's training data. We sample an execution of ANON using the workload generated by a developer-written integration test suite. This workload exercises important program states involving objects essential for testing ANON. The sampled execution yields 840 unique objects serialized from 62 classes. Row ANON in Table 1 reports the statistics of these serialized objects. Section 4.5 reports the results of evaluating X2J on ANON that we conducted with proprietary code (which we are not allowed to include in JObjectsBench) to confirm the validity of the results on code that has certainly not been used to train the LLMs.

Baselines. We compare X2J with two baselines, ProDJ [58] and a set of vanilla LLM approaches. ProDJ is a technique for capturing object states at runtime and generating code to reproduce the captured objects. ProDJ can be used in two configuration modes: structure-based and trace-based. The structure-based mode synthesizes a construction plan

from the static structure of the target class and instruments the program to record the concrete values that populate it. The trace-based mode is semi-automated, since it requires per-project instrumentation, selection of tracing targets and manual configuration of target workloads. We configure ProDJ for each benchmark to ensure methodological fairness across our evaluation. We execute both approaches on the ProDJ benchmark under identical workloads to assess their full capabilities. We restrict the comparison on the JObjectsBench benchmark to ProDJ structure-based variant (ProDJ$_{SB}$) as ProDJ lacks a flexible runtime tracing mechanism, requiring project-specific configuration effort to adapt to diverse target codebases. Therefore, the results of ProDJ$_{SB}$ on the JObjectsBench benchmark reveal only the capability of structure-based reconstruction, serving as a baseline that combines static code analysis with field values captured at runtime. [6]

The vanilla LLM approaches prompt an LLM with the object reconstruction task and the object serialized in XML format. The prompt does not include any source code from the Java project to facilitate the reconstruction task (X2J does). We use this baseline approach with six open-source LLMs (DeepSeek-R1-32B [13], GPT-OSS-20B [42], Gemma-3-27B [51], Llama-3.1-70B [36], Qwen2.5-Coder-32B [26], Qwen3-Coder-30B [54]) and report the results for all of them. We select these LLMs as they are available on `Ollama`[7] and can run on a local cluster. We decided to use `Ollama` as we needed a local infrastructure to run the experiments on the closed-source application included in our evaluation (as we were required not to disclose any data). We used default model parameters and set the temperature value equal to zero for all the LLMs.

Setup. We deployed Ollama service on a cluster of four nodes to run LLMs for both embedding and inference. Each node runs Ubuntu 20.04 equipped with an NVIDIA A100 80GB PCIe GPU and 14 Intel Haswell CPUs (32 cores, 2.8 GHz). We conducted all the experiments using the cluster. We used LlamaIndex[8] for AST-Aware RAG as it provides high-level interfaces to index input chunks and computes normalized cosine similarity scores (ranging from 0 to 1) for each retrieved context chunk. We designed X2J so it can be configured to use different LLMs.

We use X2J in its default configuration, which employs Qwen3-Coder-30B as the LLM with a temperature value of 0.4. We use Qwen3-Coder-30B as the default LLM for X2J because it achieves the best accuracy as reported in Section 4.2. We selected the temperature value, with a set of preliminary experiments on a 5% random sample of objects from JObjectsBench. We measure the accuracy of the approach (number of correctly recreated objects over the total number of objects) using 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0 (0.2 increments from 0.0 to 1.0) as temperature values for the LLM. The accuracy remains stable between 0.0 and 0.8, but drops significantly when the temperature is equal to 1.0. The maximum accuracy is obtained when a temperature of 0.4 that we use as the default value when evaluating X2J. We investigate the impact of context windows that exceed the context limits of standard LLM configurations on object reconstruction by experimenting with a 64K-context configuration (X2J$_{64K}$) on the ProDJ benchmark.

4.2 RQ1: Effectiveness

We assess the effectiveness of the approach in terms of accuracy, that is, the proportion of *exactly* reconstructed objects out of all objects. We compare the object graph of a reconstructed object with its original counterpart to determine whether it is exactly reconstructed. As detailed in Section 3, we obtain the object graph by serializing runtime objects into XML format using XStream and compare the resulting XML documents using DeepDiff, that checks if two XML

[6] We were not able to compare with CoDeSe [24] (a seminal approach for generating code reproducing the state of Java objects), since we have not been able to access the tool that traces back to 2011.
[7] https://ollama.com/
[8] https://www.llamaindex.ai/

Table 2. Accuracy (%) of reconstructed objects (DS-32B: DeepSeek-R1-32B, GO-20B: GPT-OSS-20B, QC-32B: Qwen2.5-Coder-32B, GM-27B: Gemma-3-27B, QC-30B: Qwen3-Coder-30B, LM-70B: Llama-3.1-70B. The *Total* row reports mean ± standard deviation)

	Vanilla LLMs						ProDJ$_{SB}$	X2J					
Project	**DS-32B**	**GO-20B**	**QC-32B**	**GM-27B**	**QC-30B**	**LM-70B**		**DS-32B**	**GO-20B**	**QC-32B**	**GM-27B**	**QC-30B**	**LM-70B**
Chart	9.4	18.4	26.1	10.4	25.0	16.9	0.1	58.8	67.1	61.7	57.1	70.7	55.6
Cli	3.3	5.2	34.6	2.0	7.2	5.2	0.0	81.0	85.6	73.2	47.7	85.0	69.9
Closure	16.2	23.4	8.1	2.7	2.7	2.7	2.7	44.1	45.9	44.1	35.1	55.9	31.5
Codec	1.5	6.0	3.0	0.0	3.0	4.5	0.0	6.0	92.5	6.0	64.2	64.2	6.0
Collections	20.9	20.9	37.3	9.0	48.9	29.9	48.9	51.8	69.8	60.8	46.0	72.7	44.1
Compress	11.2	12.1	9.3	9.3	12.1	12.1	0.0	52.3	23.4	50.5	48.6	60.7	40.2
Csv	0.0	6.1	4.5	0.0	6.1	3.0	4.5	65.2	87.9	83.3	31.8	83.3	57.6
Gson	20.7	38.3	21.6	22.5	30.6	20.3	0.5	61.3	75.2	64.0	44.6	72.5	61.7
JacksonCore	1.7	8.8	5.0	3.8	4.2	3.8	0.4	40.0	35.0	59.2	41.7	81.2	65.8
JacksonDatabind	2.7	30.9	28.2	7.3	43.2	2.7	0.0	40.9	79.1	55.5	45.0	67.3	53.2
JacksonXml	0.0	50.0	0.0	0.0	62.5	0.0	0.0	75.0	100.0	75.0	43.8	81.2	100.0
Jsoup	7.5	12.8	12.0	10.5	12.0	15.0	6.8	22.6	21.8	29.3	37.6	44.4	23.3
JxPath	7.0	11.7	39.8	71.3	10.5	10.5	0.0	76.6	76.6	77.2	52.0	75.4	76.0
Lang	15.2	20.4	23.9	12.5	15.2	16.3	2.1	70.6	77.9	79.9	64.7	81.3	48.1
Math	44.3	55.2	42.7	39.2	51.9	43.9	0.0	67.1	69.7	69.9	53.2	75.4	62.0
Mockito	14.7	23.5	37.3	19.6	39.2	41.2	0.0	51.0	51.0	51.0	49.0	51.0	49.0
Time	8.8	13.2	14.6	59.4	11.9	54.1	0.4	83.8	85.6	85.9	78.7	89.0	77.1
Total	16.1 ± 11.0	24.3 ± 14.9	25.5 ± 14.4	26.1 ± 21.0	26.1 ± 19.5	27.2 ± 16.4	3.6 ± 11.7	62.6 ± 20.8	69.8 ± 23.7	67.0 ± 20.5	56.2 ± 11.7	75.1 ± 12.6	59.0 ± 22.0

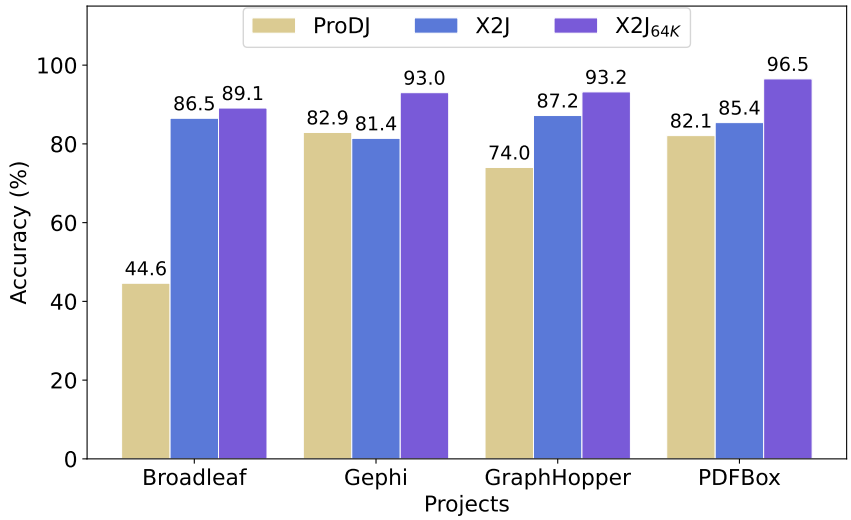


Fig. 3. Comparison of object reconstruction accuracy among ProDJ, X2J, and X2J$_{64K}$ on the ProDJ benchmark

files have identical values and attributes. Table 2 reports the accuracy of X2J configured with different LLMs and the baselines, ProDJ$_{SB}$ and the vanilla LLMs.

Comparison with ProDJ. The accuracy of ProDJ$_{SB}$ (col. ProDJ$_{SB}$) on the JObjectsBench benchmark is nearly zero on all projects except for `Collections` (48.9%), with an overall average of 3.6%. The low accuracy of ProDJ$_{SB}$ highlights the limitations of rule-based approaches that can effectively handle only objects that match the rule set. ProDJ$_{SB}$ relies on static code analysis to find corresponding constructors, getters and setters that are able to reconstruct object states. Thus, it often fails to generate correct code when the project's structure is complex. Most projects in JObjectsBench contain a broad variety of custom domain classes that do not match the ProDJ$_{SB}$ rules. The intricate inheritance hierarchies and deeply nested structures described in Section 4.1 further complicate object reconstruction

when relying solely on class-code analysis. Only the `Collections` project, which contains many objects of a relatively small set of types, such as variations of Java collections, including `Set`, `List`, and `Map`, aligns well with the rule-based approach of ProDJ$_{SB}$.

X2J achieves 85.4% average accuracy, and its scaled-up 64K-context variant (X2J$_{64K}$) reaches 92.0% under the same experimental conditions on ProDJ benchmark. Both consistently outperform ProDJ 70.9% average accuracy, as shown in Figure 3. The largest relative improvement (+93.9%) of X2J (86.5%) on ProDJ (44.6%) for `Broadleaf` depends on the complex dependency injection and dynamic proxy generation that exceed the static rule-based template. ProDJ slightly outperforms the default X2J (82.9% vs. 81.4%) on `Gephi`, since `Gephi` heavily relies on parameterized constructor calls, which ProDJ's tracing mechanism captures natively. These results indicate that X2J generalizes more robustly than ProDJ, particularly as contextual scope expands.

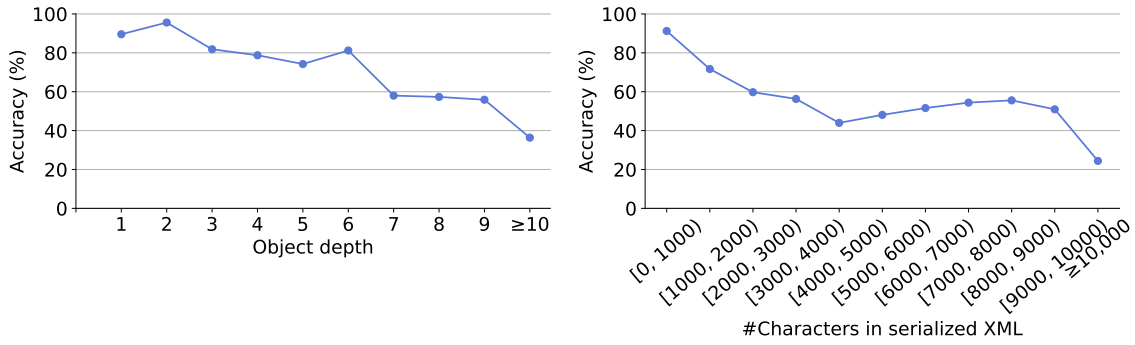
Comparison with Vanilla LLMs. We configure X2J with six LLMs and compare against their vanilla settings on the JObjectsBench benchmark. The accuracy of the vanilla models (col. *Vanilla LLMs*) ranges between 16.1 and 27.2 with the standard deviation ranging from 11.0 to 21.0, suggesting they have similar behavior across all projects. The size of the LLMs does not significantly impact the accuracy of the models: Llama-3.1-70B, the largest model in the set, reaches the highest average accuracy (col. *LM-70B*) among the vanilla group (27.2%); however, 30B-class models are close behind it and sometimes perform better than the larger model. This is the case for Gemma-3-27B (col. *GM-27B*) on `JxPath` and Qwen3-Coder-30B (col. *QC-30B*) on `JacksonXml`. Compared to ProDJ$_{SB}$, all vanilla LLMs perform better, except in the `Collection` project, where the rule-based approach outperforms all but one (Qwen-3-Coder-30B) of them. The low accuracy of vanilla LLMs underscores the necessity of a more advanced approach for object reconstruction.

We experimented with six LLMs to investigate the impact of X2J on different vanilla LLMs. The six different configurations of X2J (col. *X2J*) perform consistently better than their vanilla settings. X2J, when configured with Llama-3.1-70B (col. *LM-70B*), DeepSeek-R1-32B (col. *DS-32B*), and Qwen3-Coder-30B (col. *QC-30B*), improves reconstruction accuracy by 116.9%, 288.8%, 187.7% respectively. X2J configured with Gemma-3-27B (col. *GM-27B*) improves reconstruction accuracy by at least 115.3%, the smallest improvement among the six LLMs considered. X2J with Qwen3-Coder-30B significantly outperforms all vanilla LLMs on *all* projects with an overall average of 75.1%. X2J shows stable results across all projects (std. 12.6). X2J largely outperforms also ProDJ$_{SB}$ across all code-bases, including `Collections`, where ProDJ$_{SB}$ performs best, suggesting that our validation-guided, LLM-driven object synthesis generalizes beyond rule-based approaches.

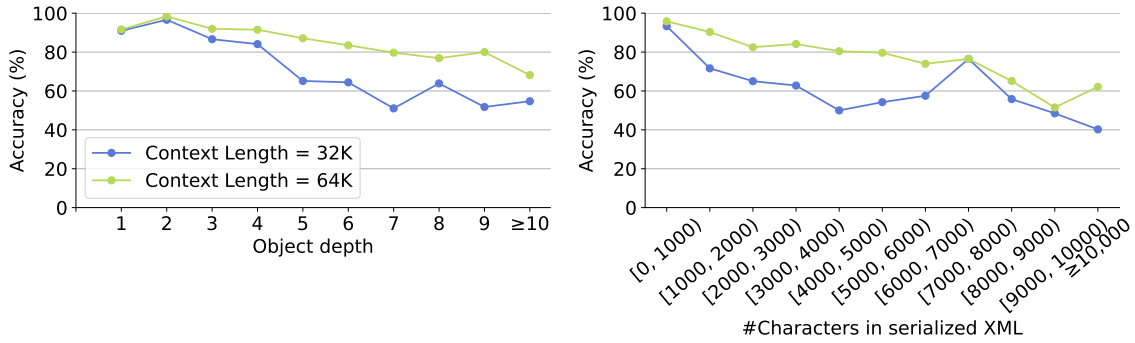
Impact of Object Complexity on Effectiveness. Figure 4a shows the accuracy of X2J for the objects grouped by depth[9] on the left-hand side, and by size of the serialized XML on the right-hand side. The figure shows a clear correlation between complexity and accuracy.

While nested object hierarchies are common in the JObjectsBench benchmark, 90.9% of instances have $depth \geq 2$, high-complexity nesting is rare, only 6.1% of instances have $depth \geq 10$. Figure 4a shows that X2J maintains strong effectiveness across the shallow-to-moderate depths that characterize most objects in the benchmark ($depth < 7$), with over 74% accuracy and a peak of 95.6%. The figure shows that accuracy decreases for the small fraction of deeply nested objects, from 58.0% at $depth = 7$ to 36.4% for $depth \geq 10$, indicating that extreme structural complexity remains challenging. X2J successfully reconstructs objects with depths as high as 24 as reported in Table 1 (the deepest object

[9]We define depth as the number of nesting levels, following Milanova's definition [38], as extended for Java objects by Wachter et al. [58]



(a) Accuracy of objects reconstructed from both JObjectsBench and ProDJ benchmarks



(b) Accuracy of objects reconstructed from ProDJ benchmark

Fig. 4. Object reconstruction accuracies for various object depths and sizes of serialized XML

in JObjectsBench reaches a depth of 35). Overall, these results show that X2J performs consistently well on the vast majority of nested hierarchies and degrades mainly on the more rare, highly nested cases.

A parallel trend emerges with serialized XML size, where the accuracy drops from 91.2% ($size < 1{,}000$) to 71.7% ($size \in [1{,}000, 2{,}000)$), 59.8% ($size \in [2{,}000, 3{,}000)$), and 56.3% ($size \in [3{,}000, 4{,}000)$). Even on the extremely long-tail instances ($size \geq 10{,}000$ characters), X2J successfully reconstructs 24.4% of objects via Validation-Driven Repair. Crucially, these reconstructions preserve structural invariants and yield compilable object reconstruction code, thus showing that X2J retains strong engineering utility under severe complexity constraints.

We observe an increased reconstruction accuracy for object depths in the range $[5, 6]$ and XML sizes in the range $[7{,}000, 9{,}000]$. Our manual inspection reveals that the majority of exactly reconstructed objects belong to complex, utility-heavy data classes. For instance, we note precise object reconstruction of the class `FastDatePrinter` from project `Lang`, the class `ObjectWriter` from project `JacksonDatabind`, and the class `DateTimeFormatter` from project `Time`. These findings highlight X2J with particular strength in accurately reconstructing objects of complex, deeply nested utility classes that typically involve intricate configuration hierarchies. The results confirm that the X2J iterative validation and self-repair loop can restore the correct state even in the presence of complex nested structures.

The experimental results indicate that expanding the context window tends to improve the effectiveness of X2J. Figure 4b shows that the reconstruction accuracy declines predictably as the object complexity approaches the LLM

Table 3. Accuracy (%) and absolute percentage improvement (Δ) of exactly reconstructed objects for ablation of X2J components ($\mathcal{T}$ - Text-based RAG, $\mathcal{A}$ - AST-Aware RAG, $\mathcal{R}$ - Validation-Driven Repair).

Project	LLM	LLM + $\mathcal{T}$		LLM + $\mathcal{A}$		LLM + $\mathcal{R}$		LLM + $\mathcal{A}$ + $\mathcal{R}$	(X2J)
	Accuracy	Accuracy	Δ	Accuracy	Δ	Accuracy	Δ	Accuracy	Δ
Chart	25.0	23.9	-1.1	31.0	+6.0	49.7	+24.7	70.7	+45.7
Cli	7.2	2.0	-5.2	37.3	+30.1	4.6	-2.6	85.0	+77.8
Closure	2.7	5.4	+2.7	26.1	+23.4	12.6	+9.9	55.9	+53.2
Codec	3.0	3.0	+0.0	6.0	+3.0	4.5	+1.5	64.2	+61.2
Collections	48.9	0.0	-48.9	49.8	+1.0	71.1	+22.2	72.7	+23.8
Compress	12.1	13.1	+0.9	23.4	+11.2	15.9	+3.7	60.7	+48.6
Csv	6.1	7.6	+1.5	34.8	+28.8	24.2	+18.2	83.3	+77.3
Gson	30.6	39.2	+8.6	43.7	+13.1	54.5	+23.9	72.5	+41.9
JacksonCore	4.2	5.0	+0.8	5.4	+1.3	51.2	+47.1	81.2	+77.1
JacksonDatabind	43.2	31.4	-11.8	12.7	-30.5	63.2	+20.0	67.3	+24.1
JacksonXml	62.5	68.8	+6.2	81.2	+18.8	100.0	+37.5	81.2	+18.8
Jsoup	12.0	26.3	+14.3	18.8	+6.8	65.4	+53.4	44.4	+32.3
JxPath	10.5	0.0	-10.5	74.9	+64.3	17.0	+6.4	75.4	+64.9
Lang	15.2	17.0	+1.7	21.8	+6.6	26.0	+10.7	81.3	+66.1
Math	51.9	47.4	-4.5	53.6	+1.7	61.7	+9.8	75.4	+23.5
Mockito	39.2	0.0	-39.2	51.0	+11.8	46.1	+6.9	51.0	+11.8
Time	11.9	46.4	+34.5	48.6	+36.8	80.7	+68.9	89.0	+77.1
Total	**26.1**	**27.5**	**+1.4**	**38.8**	**+12.7**	**53.9**	**+27.8**	**75.1**	**+49.0**

context window limit, reflecting the diminishing return of attention mechanisms over large state graphs. Extending the context length from 32K to 64K tokens systematically alleviates this constraint: the reconstruction accuracy for objects with nesting depth < 7 on the ProDJ benchmark remains consistently above 83%, and the accuracy for XML objects smaller than 5,000 characters stays above 80%. This scaling behavior indicates that the complexity limits are largely contextual and can be extended through standard context window configuration rather than requiring architectural modifications.

Overall, the results show that X2J achieves consistently high accuracy on the majority of objects, including shallow to moderately deep objects with small to medium size, while its performance gradually degrades with increasing object complexity. From a testing-practice perspective, the complexity range in which X2J performs best corresponds to the object states that developers are most likely to encounter: empirical studies of Java tests show that typical testing states are compact and hierarchically shallow [43]. The repair-loop allows the approach to tolerate nested structures up to depth 24 and XML objects with more than 10K characters, although exact reconstruction becomes rare in those extremes. For testing practice, this means that the objects developers most frequently need to reinstantiate (the common, non-pathological cases) are reconstructed with high fidelity, while even the hardest cases yield compilable code that developers can adapt with potentially modest manual effort.

RQ1 Findings: X2J effectively reconstructs objects and achieves substantially higher accuracy than the baselines in generating object reconstruction code.

4.3 RQ2: Ablation Study

X2J leverages several components, namely LLM, AST-Aware RAG ($\mathcal{A}$) and Validation-Driven Repair ($\mathcal{R}$). Table 3 shows the results of the ablation study on JObjectsBench that we ran to assess the contribution of these components to the accuracy of X2J. We conduct all ablation experiments using the default LLM configuration of X2J (Qwen3-Coder-30B). Column *LLM* reports the results of plain Qwen3-Coder-30B, the LLM used in X2J (reported also in Table 2 Column `Qwen3-Coder-30B`). Column *LLM* + $\mathcal{T}$ reports the results obtained by extending the LLM with standard text-based

RAG ($\mathcal{T}$). Column *LLM + $\mathcal{A}$* reports the results obtained using AST-Aware RAG only. Column *LLM + $\mathcal{R}$* reports the results achieved by extending the vanilla LLM with Validation-Driven Repair only. Column *LLM + $\mathcal{A}$ + $\mathcal{R}$* reports the results obtained by X2J (same as Column `X2J` of Table 2).

Text-based RAG ($\mathcal{T}$). X2J with a purely text-based retrieval component produces only marginal gains overall (Avg. +1.4%) with respect to the baseline LLM. It even reduces accuracy in some projects, for instance, −48.9% on `Collections`, −39.2% on `Mockito`. This instability suggests that naïve retrieval of surrounding code fragments can introduce noise into prompts rather than actionable context. Although some projects (for instance `Gson` and `Jsoup`) see modest improvements, text-based RAG alone is not sufficient to achieve reliable object reconstruction.

AST-aware RAG ($\mathcal{A}$). Replacing text retrieval with AST-aware retrieval yields a significant and consistent improvement across projects. The accuracy increases from 26.1% to 38.8% (+12.7%) on average. The gains are particularly pronounced in projects where object graphs require precise knowledge of constructors and fields: `Cli` (+30.1%), `Csv` (+28.8%), and `JxPath` (+64.3%). By focusing only on structurally relevant code elements, the AST-aware component avoids the noise observed using $\mathcal{T}$ and provides the LLM with concrete building blocks useful for object reconstruction.

Although AST-Aware RAG generally achieves high accuracy, its retrieval precision drops under certain conditions. For example, the accuracy degrades in the `JacksonDatabind` project. Our analysis reveals that this stems from a fundamental limitation in static type resolution: polymorphic instances are dynamically instantiated at runtime, while their static declarations in the source code reference only abstract base classes or interfaces. Consequently, X2J's AST-Aware RAG fails to statically relate interface-level specifications with their runtime implementations for some complex scenarios, highlighting a persistent challenge in code retrieval [64]. To mitigate this limitation, we plan to explore hybrid retrieval approaches that complement static analysis with dynamic tracing to capture runtime polymorphic behavior.

Validation-Driven Repair ($\mathcal{R}$). Extending the vanilla LLM with Validation-Driven Repair alone increases accuracy from 26.1% to 53.9% (+27.8%) on average. The improvement occurs across nearly all projects, with `Time` showing the largest improvement (+68.9%), followed by `Jsoup` (+53.4%) and `JacksonCore` (+47.1%), while `Cli` experiences a marginal decrease (−2.6%). These results indicate that Validation-Driven Repair alone can substantially improve reconstruction accuracy. However, Validation-Driven Repair does not provide the LLM with the structural context available through AST-Aware RAG: it provides validation error logs but does not retrieve the relevant code snippets needed to diagnose and correct the underlying issue. Consequently, the LLM must infer the appropriate correction from the Validation-Driven Repair alone, limiting the effectiveness of repair compared with the combination of AST-Aware RAG and Validation-Driven Repair (the full X2J approach).

Full approach ($\mathcal{A}$ + $\mathcal{R}$). The complete X2J configuration, combining AST-Aware RAG with Validation-Driven Repair, achieves the highest performance: 75.1% exact reconstruction accuracy overall (+49.0% over the plain LLM baseline). The self-repair loop proves crucial for closing the remaining gap, as it systematically validates, corrects, and refines generated code until convergence. Improvements are dramatic across all projects, with jumps of +77.8% on `Cli`, +77.3% on `Csv`, +77.1% on `Time`, and over +60% on `JxPath` and `Lang`. Validation-Driven Repair yields no accuracy gain on `Mockito`. Our manual inspection reveals a correlation between this and the reliance of `Mockito` on the dynamic binding mechanism, particularly for `Invocation`-related objects. The static context of AST-Aware RAG remains structurally insufficient to guide the repair loop toward valid corrections because these classes are dynamically instantiated and lack explicit type annotations at their call sites. These results confirm that both $\mathcal{A}$ and $\mathcal{R}$ are key

assets: the former supplies precise contextual knowledge, and the latter ensures that reconstruction is both syntactically correct and semantically faithful.

RQ2 Findings: The contribution of all components is critical to achieve the good accuracy of X2J. The ablation study confirms that both AST-Aware RAG and Validation-Driven Repair are critical, together boosting accuracy by 49.0% over the base LLM.

4.4 RQ3: Costs

We evaluate the cost of running X2J to reconstruct object states by measuring the number of consumed tokens with LLM Inference, the execution time of each phase of X2J, and the number of iterations performed by Validation-Driven Repair. The reported measurements characterize the end-to-end reconstruction cost of X2J, including both exactly reconstructed objects and failed reconstruction attempts.

Token Consumption. X2J consumed a median of 6,271 tokens to fully reconstruct an object, including the XML serialization, the initial retrieval step, and the additional steps to repair the generated code. X2J consumed from 2,210 to 10,003 tokens (median: 3,550) in the initial retrieval phase. X2J consumed from 0 to 32,130 tokens (median: 2,444) during the additional steps to repair the generated code.

Figure 6 shows the number of consumed tokens per object reconstruction categorized by project. The number of consumed tokens depends on both the kind of the project and the complexity of the objects. The median token consumption ranges from 3,300 tokens on `JacksonXml` to 14,941 tokens on `Closure`, which is consistent with Table 1. `Closure` has the highest average and maximum number of XML characters per object. `Math` is an outlier in the tail of the distribution rather than in the median (5,518 tokens). 95% of the objects consumed less than 25K tokens in the `Math` project. Although `Math` objects are smaller than `Closure` or `Chart` objects on average, `Math` contains algebraic types (for example, matrices, polynomials, and statistical estimators) that the LLM occasionally misunderstands on the first pass, triggering Validation-Driven Repair that inflates the worst-case token cost. The number of consumed tokens remains consistently below 30,000 on the other projects, reflecting the same observation: most repair iterations terminate within few iterations when the underlying types are common Java idioms.

Execution Time. We measure end-to-end execution time of X2J for reconstructing objects, including AST-Aware RAG, LLM Inference and Validation-Driven Repair. We recorded the execution time of X2J on our local GPU cluster that we describe in Section 4.1. X2J took a median of 33 seconds to reconstruct an object and approximately 90% of cases completed within one minute. Figure 5 shows that X2J spends most of the time on LLM Inference (median: 14 seconds) and Validation-Driven Repair (median: 15 seconds). The LLM Inference phase has a median runtime of 14 seconds (P_{50}, median), extending to 193 seconds at P_{90} (90^{th} percentile). The Validation-Driven Repair phase took 25 seconds per object on average, reaching 53 seconds at P_{90}, as it needs to compile and execute the generated code. AST-Aware RAG took almost negligible time (median: 1 second) to retrieve related code context.

While the vast majority of objects benefit from this rapid turnaround, a small fraction has significantly longer latencies (up to 30 minutes for LLM Inference and 491 seconds for Validation-Driven Repair). These tail situations are primarily due to transient scheduling overhead and occasional GPU context-switching instability in our locally managed cluster. Crucially, this runtime profile demonstrates that object reconstruction is highly efficient for typical testing workloads, with extreme delays confined to a negligible minority (< 1%) of cases that rarely appear.

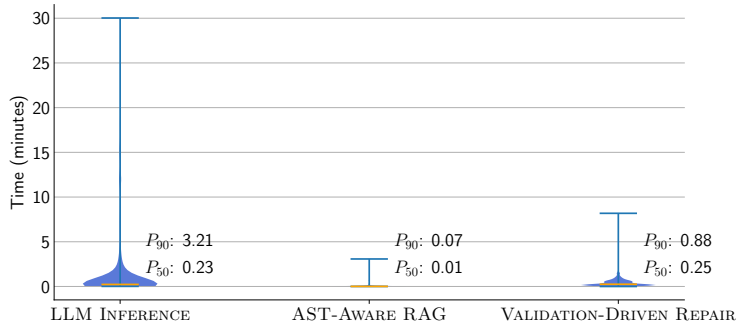


Fig. 5. Duration of each phase of X2J per object reconstruction. P_{50} (median) and P_{90} (90^{th} percentile) markers denote execution efficiency

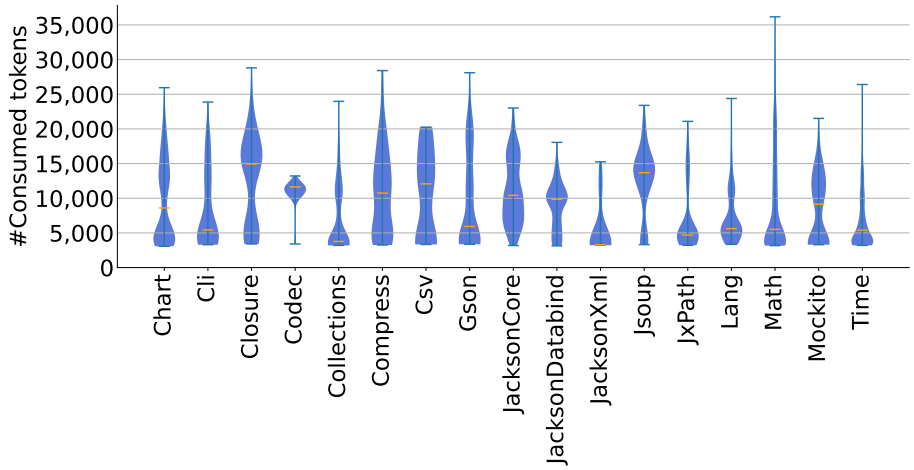


Fig. 6. Number of consumed tokens by the LLM per object reconstruction (orange lines represent medians).

Repair Iterations. A successful object reconstruction undergoes a four-step validation pipeline, which includes parsing, compilation, execution, and equivalence checking (see Section 3.4). X2J successfully reconstructed objects with a median of two iterations in the Validation-Driven Repair phase. The correct reconstruction of objects required six iterations in only 10% of the cases. X2J did not need any repair (one iteration) for 56.6% of correctly reconstructed objects. It required an iteration for correcting some syntax issues for 2.5% of correctly reconstructed objects. It required two or more iterations for fixing compilation errors (for instance unresolved symbols) for 36.4% of correctly reconstructed objects. It required an iteration for correcting some execution issues for 5.5% of correctly reconstructed objects, and two or more iterations to fix the object graphs for 10.6% of correctly reconstructed objects. The repair steps failed to repair a repeated error in less than 5% of the cases.

```
1 Option option = new Option(null, "block-size", false,
        "use SIZE-byte blocks");
2 option.setRequired(false);
3 option.setOptionalArg(false);
4 option.setArgs(1);
5 option.getValuesList().add("10");
6 option.setValueSeparator('=');
7 CommandLine line = new CommandLine.Builder().addOption
        (option).build();
```

Listing 5. Code generated with X2J to create the object

```
1 CommandLineParser parser = new PosixParser();
2 Options options = new Options();
3 options.addOption(OptionBuilder.withLongOpt( "block-
        size" ).withDescription( "use SIZE-byte blocks" )
        .withValueSeparator( '=' ).hasArg().create());
4 String[] args = new String[]{ "--block-size=10" };
5 CommandLine line = parser.parse( options, args );
```

Listing 6. Original source code to create the `line` object

RQ3 Findings: X2J is cost-effective. It consumes a reasonable number of tokens, compatible with the use of local LLMs. It needs a limited number of iterations in its iterative VALIDATION-DRIVEN REPAIR step, and generally incurs a negligible waiting time to complete.

4.5 RQ4: Practical Usefulness

We evaluate the practical usefulness of X2J by estimating the quality of the produced code, the ability to reproduce bugs, and the industrial applicability.

Code Quality. We measure the quality of X2J in terms of the quality of the objects that it correctly reconstructs in JOBJECTSBENCH benchmark (3,723 objects with extractable ground-truth code). We measure the quality of the generated objects by comparing the number of lines of code, the number of method calls, and the cognitive complexity score of the generated object with the corresponding code in the original object that we extract with static analysis.

Listings 5 and 6 show the code that X2J produces to generate the object and the original source code of project `Cli` that creates the object, respectively. The two code snippets generate the same `CommandLine` object (based on its serialized representation) using related but slightly different statements: X2J code (Listing 5) reconstructs a `CommandLine` object with 7 lines of code, 9 method calls, and a cognitive complexity of 0; the original code (listing 6) generates the object with 5 lines of code, 9 method calls, and a cognitive complexity of 0. Figure 7 illustrates the distribution of the three metrics for both generated (*X2J*) and original (*Ground truth*) code for creating the objects.

X2J generates a median of 5 lines of code to reconstruct an object state, less than the ground truth code (median: 11) that creates the object. In the best case, X2J generates 3 lines of code, the same as the ground truth, while in the worst case, X2J generates 131 lines of code, 25 lines more than the ground truth code. The number of method calls shows a similar pattern with an average of 3 calls for the generated code and an average of 5 calls for the ground truth. The ground truth code calls up to 187 methods, while X2J invokes at most 124 methods. This is due to different patterns used in the code. To create complex objects, a developer may prefer to separate a method call with complex arguments into multiple segments, while the LLM may directly generate the code with all parameters properly instantiated.

The cognitive complexity is a widely used metric that Sonar[10] proposes to measure the understandability of the methods. X2J generates code with a desirable low cognitive complexity, with almost no uselessly nested structures or redundant complex programming constructs. The right-most violin plot in Figure 7 shows a similar distribution of cognitive complexity between X2J and the ground truth code, in both cases consistently below the production threshold (15), thus showing that X2J can generate production-acceptable code with good understandability.

[10] https://www.sonarsource.com/resources/cognitive-complexity/

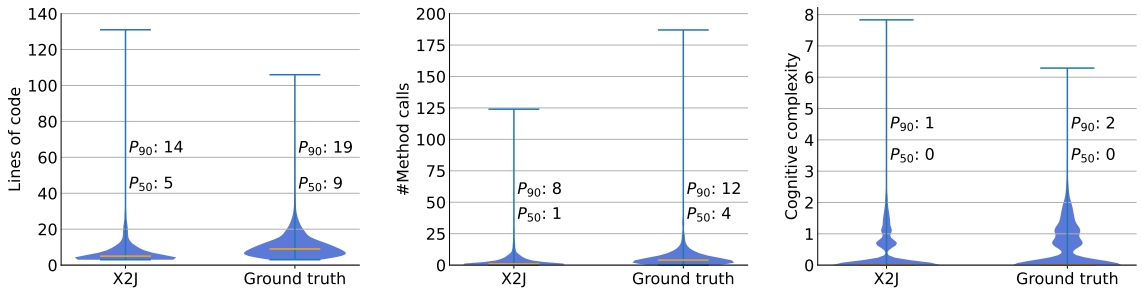


Fig. 7. Code quality metrics per reconstructed object (orange lines represent medians). P_{50} (median) and P_{90} (90^{th} percentile) markers denote the quality distribution across the evaluated objects.

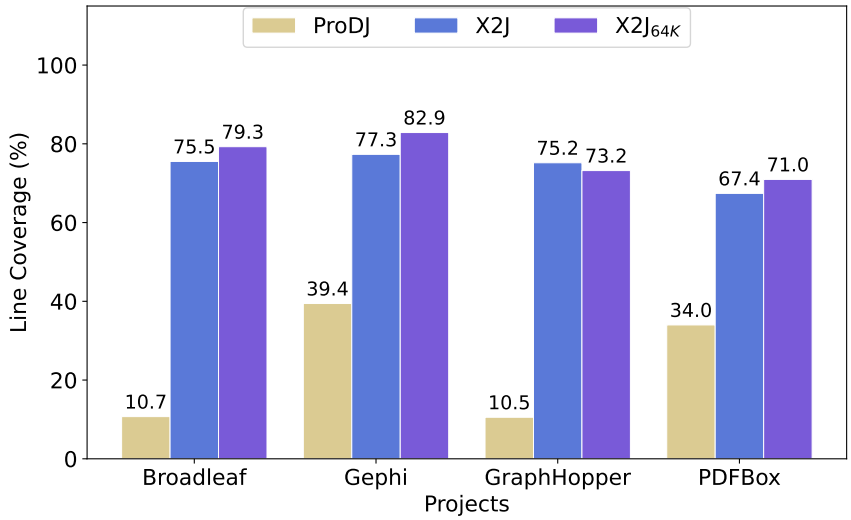


Fig. 8. Comparison of line coverage between ProDJ, X2J, and its scaled-up variant with a context length of 64K tokens (X2J$_{64K}$) on the ProDJ benchmark.

In a nutshell, our experiments show that X2J generates human-readable code that is consistent with the code written by the developers.

Test Effectiveness. We quantify the impact of object reconstruction on the effectiveness of testing by computing the line and branch coverage when replaying the ProDJ benchmark. We first execute the original workload to obtain the set of executed lines and branches, then we execute the generated tests and measure the fraction of the executed lines and branches that are also present in the baseline workload execution. Thus, the reported coverage represents the fraction of the behavior exercised by the original workload that is reproduced by the generated tests, rather than coverage of the entire codebase. Figure 8 and Figure 9 show the comparative coverage results for ProDJ, X2J, and X2J$_{64K}$ across the ProDJ benchmark. Overall, X2J achieves an average line coverage of 73.9% (ranging from 67.4% to 77.3%) and branch coverage of 37.7% (30.4% – 50.9%). ProDJ achieves 23.7% line coverage and 17.3% branch coverage on average. We observe the largest gap between X2J and ProDJ for Broadleaf (+64.8 percentage points in line coverage

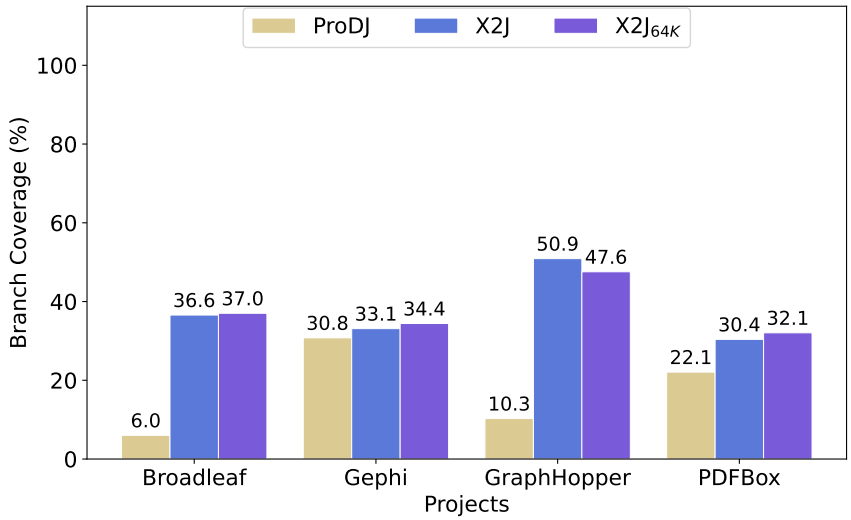


Fig. 9. Comparison of branch coverage between ProDJ, X2J, and its scaled-up variant with a context length of 64K tokens ($X2J_{64K}$) on the ProDJ benchmark

and +30.5 percentage points in branch coverage) and GraphHopper (+64.6 percentage points line and +40.6 percentage points branch). The low coverage of ProDJ depends on ProDJ 's reliance on static templates, which frequently fail to bind critical domain objects during replay, thereby preventing the traversal of dependent method invocation chains. The branch coverage remains bounded by the intrinsic constraints of workload replay: a single reference trace necessarily exercises only a feasible subset of control-flow paths. Consequently, X2J reproduces observed branches rather than exploring unvisited alternatives.

Bug Reproducibility. We investigate the capability of X2J to generate test cases for reproducing industrially-relevant bugs. We selected the first three bugs per project from Defects4J (e.g. Chart-1b, Chart-2b and Chart-3b), which are non-crash and directly involve at least a structured object. For each bug, we used X2J to capture all the objects needed to invoke the focal method by executing a test case that triggers the bug. We collected the objects from the states that we observed during the execution. We generated the code that reproduces the objects and verified it by executing the code, calling the method that triggers the bug, and executing the assertions from the test cases in the dataset.

Table 4 shows the number of exactly reconstructed objects, the number of generated objects (including the incorrect objects that do not reproduce the bug), and the total number of objects created when running each bug-triggering test, for the three bugs we analyzed per project. Each cell of the table reports a triple `a/b/c`, where `a` is the number of objects that X2J reconstructs *exactly* (the equivalence check passes), `b` is the number of *generated* objects (those for which X2J produces compilable code, including objects that do not pass the equivalence check), and `c` is the total number of objects *captured* during the execution of the bug-triggering test. For example, the triple `4/5/6` for Chart bug #1 reads: "X2J captures 6 objects from the bug-triggering test, generates compilable code for 5 of them, and exactly reconstructs 4." The boldface data highlight the generation of one or more objects. The light green background highlights the partially reconstructed bugs, the purple background highlights the fully reconstructed bugs. X2J successfully reproduces 17 out

Table 4. Number of reconstructed / generated / total objects per bug by X2J

Project	#Objects per Bug		
	1st Bug	2nd Bug	3rd Bug
Chart	**4/5/6**	**4/4/5**	**4/6/6**
Cli	**1/1/2**	**2/2/2**	0/0/1
Closure	**2/2/2**	**1/1/1**	0/0/1
Codec	**0/1/2**	**1/1/1**	**1/1/1**
Collections	**1/2/2**	**2/7/7**	0/0/1
Compress	**0/5/5**	**0/3/4**	**0/4/5**
Csv	**1/1/1**	**1/1/1**	0/0/1
Gson	**1/1/1**	**0/1/1**	0/0/2
JacksonCore	**2/2/3**	**2/2/2**	**3/3/3**
JacksonDatabind	**1/1/1**	**4/4/4**	0/0/3
JacksonXml	**1/1/1**	0/0/1	0/0/1
Jsoup	**0/1/1**	**0/1/1**	**0/2/2**
JxPath	0/0/4	0/0/1	0/0/1
Lang	**0/1/1**	**4/4/4**	**1/1/1**
Math	**1/1/2**	0/0/1	0/0/2
Mockito	0/0/1	0/0/1	0/0/1
Time	**1/1/1**	**2/2/2**	0/0/2

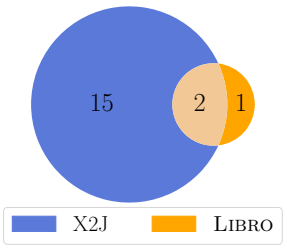


Fig. 10. Venn diagram for bug reproduction.

of 51 selected bugs. It partially reproduces eight bugs, that is, it reproduces at least one bug-related object. X2J fails to generate any objects for 17 bugs, while it generates only incorrect objects for 9 bugs.

We explored the complementarity of X2J with state-of-the-art techniques, by comparing the bugs that we reveal with test cases generated with X2J with the bugs that we reproduce with the state-of-the-art approach LIBRO [29]. LIBRO takes a natural-language bug report as input, prompts an LLM with the report and one or two few-shot examples, and produces JUnit tests that shall fail on the buggy version and pass on the fixed version. LIBRO does not perform any runtime monitoring or object capture, therefore its inputs are complementary to X2J inputs. We executed LIBRO in the best performing configuration as indicated in the documentation. Figure 10 shows the complementarity of X2J and LIBRO in detecting bugs across the sampled failure cases. Both X2J and LIBRO reproduce two shared bugs, while each also captures a distinct subset of issues that the other misses (LIBRO identifies one exclusive failure, whereas X2J detects fifteen additional cases). This partial divergence indicates that the approaches target different operational conditions and failure modes rather than operating as direct substitutes. Integrating both techniques into a comprehensive testing workflow would broaden debugging of failures and regression testing capabilities.

Industrial Applicability. We evaluated the industrial applicability of X2J, by deploying the X2J pipeline to reconstruct objects for an industrial Java web application (ANON) that is implemented with the Spring Boot framework.[11] We enabled full continuous integration (CI) pipeline access for ANON, allowing the complete X2J pipeline, including AST-AWARE RAG and VALIDATION-DRIVEN REPAIR, to be executed end-to-end on the industrial system. We maintained identical configuration settings and the same base LLM (Qwen3-Coder-30B) as used in our JOBJECTSBENCH and PRODJ benchmark experiments. The target application is proprietary and deployed in production. The source code of the application is kept private and, therefore, we are confident that the application is not part of the LLM's training set, ensuring that our evaluation strictly measures object reconstruction performance beyond memorized benchmarks. We validated X2J on a sampled execution of ANON, where OBJECT CAPTURE serialized 840 distinct objects from 62 classes. Row ANON in Table 1 shows the statistics of the captured objects.

[11] https://github.com/spring-projects/spring-boot

X2J successfully reconstructed 752 of the 840 objects, achieving an accuracy of 89.5%, in line with the average accuracy observed across our benchmark datasets (80.7%). We observed that 49.2% of reconstruction scenarios triggered the Validation-Driven Repair phase, requiring further refinement of object reconstruction code. Among all the exactly reconstructed scenarios, 49.2% of scenarios required fixing compilation issues in the generated code, while fewer than 5% required additional iterative refinement to resolve syntax errors, execution failures, or object state inconsistencies. These results confirm that our results on publicly available benchmarks do not suffer from training biases, and that X2J can effectively handle the complexity and variability of industrial-scale Java projects, demonstrating strong applicability in industrial environments.

RQ4 Findings: X2J is applicable in practice: it generates code with quality similar to the original code extracted from developer-written tests, provides strong capability to reproduce real-world bugs involving multiple objects, and can reproduce objects for a proprietary industrial web application in production.

4.6 Threats to Validity

Internal Validity. The threats to the internal validity of our results come from the stochasticity of LLM outputs, the limited iterations in the Validation-Driven Repair phase, and the criterion used to determine exact reconstruction.

The responses of the LLMs are not deterministic. We mitigate this threat by deploying local LLMs with the default configurations, sampling different temperatures, and verifying the performance with more than ten thousand objects. We stop the Validation-Driven Repair phase after six iterations. Some failed objects might succeed with a higher limit. Thus, the results we report in the paper are conservative. We determine whether a reconstruction is exact by comparing the XStream XML serializations of the reconstructed and captured object graphs. This criterion provides an objective and reproducible oracle and directly compares the state observed by the serializer. However, it is stricter than program-level semantic equivalence: two objects may be semantically equivalent while differing in fields that do not affect the behavior relevant to the application. In our evaluation, such cases are counted as inexact. Therefore, the results we report in the paper are conservative.

The overall accuracy of 80.7% across 10,794 objects from 21 Java projects indicates that X2J should be regarded as a developer support tool rather than a fully autonomous reconstruction mechanism. The equivalence-checking step in Validation-Driven Repair is therefore important for practical use: it exposes serialization differences and can reduce the developer's review effort to inspecting localized differences in the XML representation. Furthermore, our experiments with a larger LLM context size ($X2J_{64K}$) indicate a possible direction for improving reconstruction accuracy.

External Validity. The main threat to the external validity of our results is the limited size and scope of the experiments. We experimented with Java projects, we used XStream to serialize the objects, and a set of currently available LLMs. The future progress of serialization libraries and LLMs may improve the results reported in this paper. The results in the paper may not generalize to other programming languages and serialization formats. We mitigate these threats by experimenting with a popular programming language, the most common serialization library, and open-source LLMs. Our ablation study uses only Qwen3-Coder-30B, the default LLM configuration of X2J. Therefore, the relative contribution of X2J's components may not generalize to other LLMs. However, RQ1 evaluates the complete X2J pipeline and the corresponding vanilla LLMs across six LLMs, providing evidence that the overall benefits of X2J are consistent across models. We experimented with bugs from the Defects4J dataset, likely part of the training set of many LLMs.

However, X2J focuses on object reconstruction, which differs from the traditional use of the Defects4J dataset for tasks such as code completion or bug fixing. In those settings, the solutions to the problems are included in the LLM's training data. By contrast, X2J operates on serialized objects in XML format, a representation that is unlikely to appear in the LLM's training corpus. Consequently, the risk of data leakage is negligible for the object-reconstruction task. Additionally, we mitigated the biases of publicly available datasets by complementing the experiments of open-source projects with experiments on a closed-source industrial Spring Boot application, where X2J achieves 89.5% reconstruction accuracy on 840 unique objects. The developers of the Spring Boot application guaranteed to us that the code was never publicly released to the training data used for LLMs. The similar accuracy on open-source and proprietary code indicates that our results come from the X2J pipeline, not from memorization of Defects4J during training.

5 Related Work

5.1 Object Serialization for Software Testing

Many common test generation techniques embed object serialization; however, they encode the object states in opaque formats that hinder test readability. In general, most approaches to serialize objects prioritize performance over readability. For example, CoDeSe generates specialized Java code and uses low-level operations to reconstruct objects, with an over 6× deserialization speedup compared to the standard Java library [24]. CoDeSe bypasses constructors and access controls, and allocates objects and sets fields via direct memory writes to minimize the serialization time at the cost of a reduced readability of the code. ReCrash records the state of the program at the failure points, and generates JUnit test cases that load state traces from external files produced during the execution [7]. The generated tests include a deserialized state from a binary 'shadow' trace file, thus ReCrash clearly emphasizes failure reproduction over code readability. PANKTI captures runtime objects and injects them into test cases as raw XML strings, resulting in test cases where important inputs appear as large inlined XML blobs that are hard to interpret [55]. OCAT augments Randoop-generated tests with calls to custom deserialization routines [27]: It stores the object data separately and brings them into the test via helper calls, which hide the intent of the test. Other systems "carve" objects from app executions and serialize them to semi-structured formats, such as JSON and XML. For example, Meta's TestGen [6] generates Kotlin tests by deserializing JSON assets at runtime to obtain complex inputs, whereas Elbaum et al. [17] deserialize XML assets to generate differential unit tests in Java. Recently, Wachter et al. proposed a plain-code serialization framework (i.e., ProDJ) that combines static analysis with dynamic tracing to capture object states and deserialize back to Java code. The experimental comparison that we discuss in Section 4.1 and summarize in Table 2 confirms the limited accuracy of ProDJ.

By prioritizing performance over readability, these approaches effectively capture the program states; however, they produce test cases that are not self-contained and often include unreadable data fixtures. X2J produces readable test cases without significantly impacting the performance, by using an LLM to directly convert the object state captured in XML format into human-readable Java code. X2J generates plain Java code that can be directly used as unit test prefix, rather than opaque binary or serialized data that need further processing. By generating human-readable object reconstruction code, X2J improves developer comprehension and makes the resulting tests easier to understand and maintain than those produced by prior serialization-based test generation methods.

5.2 LLM-Based Test Generation

Alagarsamy et al. demonstrated that fine-tuned LLMs can generate JUnit tests with higher coverage and correctness than untuned models [3], while Fu et al. revealed that LLM-generated code often contains security weaknesses, motivating the need for automated mitigation strategies based on software testing [21]. Our approach focuses not on generating tests directly but on reconstructing realistic program states from production data to enable more realistic testing without fine-tuning a model. Lemieux et al. proposed CodaMOSA, which hybridizes search-based testing with LLM generation to overcome coverage plateaus [31], and Ahmed et al. showed that augmenting prompts with semantic code facts improves summarization and completion quality [2]. By contrast, X2J targets object-state reconstruction, where AST-aware retrieval and validation-driven repair are critical for handling complex object structures. Schäfer et al. conducted a large-scale empirical evaluation of LLM-generated JavaScript unit tests and proposed TestPilot, an iterative approach, achieving significantly higher coverage than state-of-the-art automated tools [48]. Mu et al. introduced ClarifyGPT, where models ask clarification questions before generating code, improving accuracy [40], while Ryan et al. developed SymPrompt, a coverage-guided prompting strategy that structures test generation by code paths to increase thoroughness [47]. In contrast, our framework integrates retrieval-augmented reasoning and automatic iterative repair to ensure the semantic fidelity of reconstructed states rather than refining prompts for test coverage.

Beyond generating new tests, Alshahwan et al. presented TestGen-LLM, which augments human-written tests at Meta, showing practical deployment with high acceptance rates [5]. Our work differs in that we generate new, executable Java code to recreate complex runtime states, reducing developer effort beyond incremental test improvements. Complementary work has advanced retrieval-augmented generation: Lewis et al. introduced the original RAG framework combining retrieval with generation [32]; Zhang et al. proposed reference-based RAG to generate unit tests with task-specific context from pre-existing tests of other methods [67]. Du et al. developed CodeGRAG, incorporating program graphs into retrieval for better code generation [16]; and Yu et al. proposed ProCC, a multi-retrieval prompting method that improves code completion [65]. Unlike these RAG-based approaches that enhance generation at the code or document level, X2J uses AST-aware RAG specifically to reconstruct object graphs into compilable, human-readable test code. Kang et al. introduced Libro, which leverages LLMs to generate bug-reproducing tests from natural language reports, demonstrating the feasibility of LLMs as "few-shot testers" [29]. Our approach goes further by enabling object reconstruction from real executions, allowing X2J to reproduce bugs from serialized states more effectively than Libro.

X2J complements the rapidly evolving landscape of LLM-based Automated Program Repair (APR). While conventional APR targets functional fault localization and patch synthesis, X2J repurposes these LLM-driven code synthesis paradigms to reconstruct complex object states as executable test scaffolding. Recent studies demonstrate that directly applying modern LLMs for code generation significantly outperforms traditional template-based methods when augmented with program-specific context [62, 66]. X2J operationalizes this insight by using our AST-Aware RAG component to precisely ground the model in relevant constructors, fields, and method signatures, thereby eliminating hallucinations common in unconstrained code synthesis. Furthermore, the Validation-Driven Repair in X2J directly mirrors the feedback-guided refinement strategies employed in autonomous APR agents [10], where generated artifacts are continuously compiled, executed, and structurally verified until convergence. However, rather than optimizing for functional fault recovery, X2J focuses on reconstructing object states exactly, positioning X2J as a complementary pipeline that bridges automated program repair methodologies with production-aware test case generation.

6 Conclusion

In this paper, we propose X2J, for supporting the generation of test cases that exercise code with objects observed at runtime. X2J generates test data from the objects serialized into a standard XML format, which it efficiently deserializes to obtain the Java code that reproduces the observed execution state. Our evaluation on JObjectsBench and the ProDJ benchmark comprising 10,794 unique objects shows that X2J achieves an average accuracy of 80.7%, largely better than both the state-of-the-art template-based approach ProDJ and vanilla LLMs. Our evaluation indicates that X2J generates human-readable code at acceptable costs. In a nutshell, X2J significantly reduces developers' effort for hand-crafting testing scenarios and reproducing bugs from runtime data.

Acknowledgments

This work is partly supported by the Swiss National Science Foundation under the SNF Grant 200021_215487 *A-Test Autonomic Software Testing*, by NSF grant SSF-2541053, by the Leonardo Joint Lab, and by *AI4SDV (AI-assisted methods and tools for Software-Defined Vehicle engineering automation)*, funded by the Chips Joint Undertaking and its members under grant agreement No 101329675. Co-funded by the European Union. Luca Di Grazia is supported by the Great Minds Fellowship of the University of St. Gallen. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] Iftekhar Ahmed, Aldeida Aleti, Haipeng Cai, Alexander Chatzigeorgiou, Pinjia He, Xing Hu, Mauro Pezzè, Denys Poshyvanyk, and Xin Xia. 2025. Artificial Intelligence for Software Engineering: The Journey So Far and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 119 (May 2025), 27 pages. doi:10.1145/3719006

[2] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. 2024. Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 220, 13 pages. doi:10.1145/3597503.3639183

[3] Saranya Alagarsamy, Chakkrit Tantithamthavorn, Wannita Takerngsaksiri, Chetan Arora, and Aldeida Aleti. 2025. Enhancing large language models for text-to-testcase generation. *Journal of Systems and Software* 230 (2025), 112531. doi:10.1016/j.jss.2025.112531

[4] Alsayed Algergawy, Marco Mesiti, Richi Nayak, and Gunter Saake. 2011. XML data clustering: An overview. *ACM Comput. Surv.* 43, 4, Article 25 (Oct. 2011), 41 pages. doi:10.1145/1978802.1978804

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

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

[7] Shay Artzi, Sunghun Kim, and Michael D. Ernst. 2008. ReCrash: Making Software Failures Reproducible by Preserving Object States. In *ECOOP 2008 – Object-Oriented Programming*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 542–565.

[8] Antonia Bertolino, Pietro Braione, Guglielmo De Angelis, Luca Gazzola, Fitsum Meshesha Kifetew, Leonardo Mariani, Matteo Orrù, Mauro Pezzè, Roberto Pietrantuono, Stefano Russo, and Paolo Tonella. 2022. A Survey of Field-based Testing Techniques. *ACM Comput. Surv.* 54, 5 (2022), 92:1–92:39. doi:10.1145/3447240

[9] Benyamin Beyzaei, Saghar Talebipour, Ghazal Rafiei, Nenad Medvidović, and Sam Malek. 2025. Automated Test Transfer across Android Apps using Large Language Models. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA098 (June 2025), 24 pages. doi:10.1145/3728975

[10] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 2188–2200. doi:10.1109/ICSE55347.2025.00157

[11] Zimin Chen, Yue Pan, Siyu Lu, Jiayi Xu, Claire Le Goues, Martin Monperrus, and He Ye. 2025. Prometheus: Unified Knowledge Graphs for Issue Resolution in Multilingual Codebases. arXiv:2507.19942 [cs.SE] https://arxiv.org/abs/2507.19942

[12] Gabriel Darbord, Nicolas Anquetil, Benoit Verhaeghe, and Anne Etien. 2025. A Multi-Language Tool for Generating Unit Tests from Execution Traces . In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA,

USA, 241–251. doi:10.1109/SANER64311.2025.00030

[13] DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

[14] Amirhossein Deljouyi and Andy Zaidman. 2023. Generating Understandable Unit Tests through End-to-End Test Scenario Carving. In *23rd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2023, Bogotá, Colombia, October 2-3, 2023*, Leon Moonen, Christian D. Newman, and Alessandra Gorla (Eds.). IEEE, 107–118. doi:10.1109/SCAM59687.2023.00021

[15] Giovanni Denaro, Noura El Moussa, Rahim Heydarov, Francesco Lomio, Mauro Pezzè, and Ketai Qiu. 2024. Predicting Failures of Autoscaling Distributed Applications. *Proc. ACM Softw. Eng.* 1, FSE, Article 87 (July 2024), 22 pages. doi:10.1145/3660794

[16] Kounianhua Du, Jizheng Chen, Renting Rui, Huacan Chai, Lingyue Fu, Wei Xia, Yasheng Wang, Ruiming Tang, Yong Yu, and Weinan Zhang. 2025. CodeGRAG: Bridging the Gap between Natural Language and Programming Language via Graphical Retrieval Augmented Generation. arXiv:2405.02355 [cs.SE] https://arxiv.org/abs/2405.02355

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

[18] Zhiyu Fan, Haifeng Ruan, Sergey Mechtaev, and Abhik Roychoudhury. 2024. Oracle-Guided Program Selection from Large Language Models. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 628–640. doi:10.1145/3650212.3680308

[19] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 141–152. doi:10.1145/3213846.3213869

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

[21] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. 2025. Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* (Feb. 2025). doi:10.1145/3716848 Just Accepted.

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

[23] Luca Gazzola, Leonardo Mariani, Matteo Orrú, Mauro Pezzè, and Martin Tappler. 2022. Testing Software in Production Environments with Data from the Field. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 58–69. doi:10.1109/ICST53961.2022.00017

[24] Milos Gligoric, Darko Marinov, and Sam Kamin. 2011. CoDeSe: fast deserialization via code generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 298–308. doi:10.1145/2001420.2001456

[25] Sijia Gu, Noor Nashid, and Ali Mesbah. 2025. LLM Test Generation via Iterative Hybrid Program Analysis. arXiv:2503.13580 [cs.SE] doi:10.1145/3744916.3764553

[26] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).

[27] Hyeonsu Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. 2010. OCAT: Object Capture-Based Automated Testing. In *Proceedings of the 2010 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 159–170. doi:10.1145/1831708.1831730

[28] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055

[29] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-Shot Testers: Exploring LLM-Based General Bug Reproduction. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2312–2323. doi:10.1109/ICSE48619.2023.00194

[30] Yuri Kuratov, Aydar Bulatov, Petr Anokhin, Ivan Rodkin, Dmitry Sorokin, Artyom Sorokin, and Mikhail Burtsev. 2024. BABILong: Testing the Limits of LLMs with Long Context Reasoning-in-a-Haystack. arXiv:2406.10149 [cs.CL] https://arxiv.org/abs/2406.10149

[31] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 919–931. doi:10.1109/ICSE48619.2023.00085

[32] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 793, 16 pages.

[33] Karl J. Lieberherr, Walter L. Hürsch, and Cun Xiao. 1994. Object-extending class transformations. *Form. Asp. Comput.* 6, 4 (July 1994), 391–416. doi:10.1007/BF01211306

[34] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-based seed object synthesis for search-based unit testing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*

(Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1068–1080. doi:10.1145/3468264.3468619

[35] Yujie Liu, Mingxuan Zhu, Jinhao Dong, Junzhe Yu, and Dan Hao. 2024. Compiler Bug Isolation via Enhanced Test Program Mutation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 819–830. doi:10.1145/3691620.3695074

[36] Meta. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[37] Meta. 2025. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. (2025). https://ai.meta.com/blog/llama-4-multimodal-intelligence/

[38] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41. doi:10.1145/1044834.1044835

[39] Davide Molinelli, Luca Di Grazia, Alberto Martin-Lopez, Michael D. Ernst, and Mauro Pezzè. 2025. Do LLMs Generate Useful Test Oracles? An Empirical Study with an Unbiased Dataset. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Seoul, Korea, Republic of). IEEE Press, 278–290. doi:10.1109/ASE63991.2025.00031

[40] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proc. ACM Softw. Eng.* 1, FSE, Article 103 (July 2024), 23 pages. doi:10.1145/3660810

[41] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 97, 13 pages. doi:10.1145/3597503.3639187

[42] OpenAI. 2025. gpt-oss-120b & gpt-oss-20b Model Card. arXiv:2508.10925 [cs.CL] https://arxiv.org/abs/2508.10925

[43] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. 2016. The impact of test case summaries on bug fixing performance: an empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 547–558. doi:10.1145/2884781.2884847

[44] Ketai Qiu, Luca Di Grazia, Alessio Gambi, Mattia Fazzini, and Mauro Pezzè. 2025. *X2J Object Reconstruction Replication Package*. doi:10.5281/zenodo.18018311

[45] Ketai Qiu, Luca Di Grazia, Leonardo Mariani, and Mauro Pezzè. 2026. E-Test: E'er-Improving Test Suites. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26)*. Association for Computing Machinery, New York, NY, USA, 3207–3219. doi:10.1145/3744916.3773126

[46] Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2025. Order Matters! An Empirical Study on Large Language Models' Input Order Bias in Software Fault Localization. arXiv:2412.18750 [cs.SE] https://arxiv.org/abs/2412.18750

[47] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. *Proc. ACM Softw. Eng.* 1, FSE, Article 43 (July 2024), 21 pages. doi:10.1145/3643769

[48] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. doi:10.1109/TSE.2023.3334955

[49] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 8634–8652. https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf

[50] Marius Smytzek, Martin Eberlein, Batuhan Serçe, Lars Grunske, and Andreas Zeller. 2024. Tests4Py: A Benchmark for System Testing. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 557–561. doi:10.1145/3663529.3663798

[51] Gemma Team. 2025. Gemma 3. (2025). https://goo.gle/Gemma3Report

[52] Gemini Team and et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv:2403.05530 [cs.CL] https://arxiv.org/abs/2403.05530

[53] JUnit team. 2020. JUnit 5 User Guide. https://docs.junit.org/5.6.3/user-guide/

[54] Qwen Team. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[55] Deepika Tiwari, Long Zhang, Martin Monperrus, and Benoit Baudry. 2021. Production Monitoring to Improve Test Suites. *IEEE Transactions on Reliability* 70, 4 (2021), 1698–1715. doi:10.1109/TR.2021.3101318

[56] Luca Traini, Federico Di Menna, and Vittorio Cortellessa. 2024. AI-driven Java Performance Testing: Balancing Result Quality with Testing Time. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 443–454. doi:10.1145/3691620.3695017

[57] W3C. 2008. Extensible Markup Language (XML) 1.0 (Fifth Edition). https://www.w3.org/TR/xml/. W3C Recommendation, November 26, 2008.

[58] Julian Wachter, Deepika Tiwari, Martin Monperrus, and Benoit Baudry. 2026. Serializing java objects in plain code. *Journal of Systems and Software* 234 (2026), 112721. doi:10.1016/j.jss.2025.112721

[59] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025. TestEval: Benchmarking Large Language Models for Test Case Generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, Luis

Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 3547–3562. doi:10.18653/v1/2025.findings-naacl.197

[60] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2025. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models. *ACM Trans. Softw. Eng. Methodol.* (Jan. 2025). doi:10.1145/3714461 Just Accepted.

[61] XStream. 2025. About XStream. (2025). https://x-stream.github.io/

[62] Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawendé F. Bissyandé, Claire Le Goues, and Shunfu Jin. 2026. MORepair: Teaching LLMs to Repair Code via Multi-Objective Fine-Tuning. *ACM Trans. Softw. Eng. Methodol.* 35, 2, Article 38 (Jan. 2026), 38 pages. doi:10.1145/3735129

[63] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 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* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1607–1619. doi:10.1145/3691620.3695529

[64] Yang Yang, Li Kuang, Jiakun Liu, Zhongxin Liu, Yingjie Xia, and David Lo. 2026. UniCoR: Modality Collaboration for Robust Cross-Language Hybrid Code Retrieval. arXiv:2512.10452 [cs.SE] https://arxiv.org/abs/2512.10452

[65] Tianchen Yu, Li Yuan, Liannan Lin, and Hongkui He. 2025. A Multiple Representation Transformer with Optimized Abstract Syntax Tree for Efficient Code Clone Detection. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 587–587.

[66] Yuntong Zhang, Andreea Costea, Ridwan Shariffdeen, Davin McCall, and Abhik Roychoudhury. 2025. EffFix: Efficient and Effective Repair of Pointer Manipulating Programs. *ACM Trans. Softw. Eng. Methodol.* 34, 3, Article 69 (Feb. 2025), 27 pages. doi:10.1145/3705310

[67] Zhe Zhang, Xingyu Liu, Yuanzhang Lin, Xiang Gao, Hailong Sun, and Yuan Yuan. 2025. Reference-Based Retrieval-Augmented Unit Test Generation. *ACM Trans. Softw. Eng. Methodol.* (Dec. 2025). doi:10.1145/3765758 Just Accepted.

[68] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA022 (June 2025), 23 pages. doi:10.1145/3728894