CITYWALK: Enhancing LLM-Based C++ Unit Test Generation via Project-Dependency Awareness and Language-Specific Knowledge

YUWEI ZHANG and QINGYUAN LU, Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences, China

KAI LIU, Shanghai Stock Exchange Technology Co., Ltd., China

WENSHENG DOU* and JIAXIN ZHU*†, Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences, China

LI QIAN, CHUNXI ZHANG, and ZHENG LIN, Shanghai Stock Exchange Technology Co., Ltd., China

JUN WEI*†, Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences, China

Unit testing plays a pivotal role in the software development lifecycle, as it ensures code quality. However, writing high-quality unit tests remains a time-consuming task for developers in practice. More recently, the application of large language models (LLMs) in automated unit test generation has demonstrated promising results. Existing approaches primarily focus on interpreted programming languages (e.g., Java), while mature solutions tailored to compiled programming languages like C++ are yet to be explored. The intricate language features of C++, such as pointers, templates, and virtual functions, pose particular challenges for LLMs in generating both executable and high-coverage unit tests. To tackle the aforementioned problems, this paper introduces CITYWALK, a novel LLM-based framework for C++ unit test generation. CITYWALK enhances LLMs by providing a comprehensive understanding of the dependency relationships within the project under test via program analysis. Furthermore, CITYWALK incorporates language-specific knowledge about C++ derived from project documentation and empirical observations, significantly improving the correctness of the LLM-generated unit tests. We implement CITYWALK by employing the widely popular LLM GPT-4o. The experimental results show that CITYWALK outperforms current state-of-the-art approaches on a collection of ten popular C++ projects. Our findings demonstrate the effectiveness of CITYWALK in generating high-quality C++ unit tests.

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

Additional Key Words and Phrases: Unit Test Generation, Large Language Model, Program Dependence Analysis, Language-Specific Knowledge, Retrieval-Augmented Generation

*Affiliated with Nanjing Institute of Software Technology, University of Chinese Academy of Sciences, Nanjing, China.
†Corresponding authors

Authors' Contact Information: Yuwei Zhang, zhangyuwei@iscas.ac.cn; Qingyuan Lu, luqingyuan22@otcaix.iscas.ac.cn, Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences, Beijing, China; Kai Liu, kliu@sse.com.cn, Shanghai Stock Exchange Technology Co., Ltd., Shanghai, China; Wensheng Dou, wsdou@otcaix.iscas.ac.cn; Jiaxin Zhu, zhujiaxin@otcaix.iscas.ac.cn, Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences, Beijing, China; Li Qian, lqian@sse.com.cn; Chunxi Zhang, chxzhang@sse.com.cn; Zheng Lin, zhenglin@sse.com.cn, Shanghai Stock Exchange Technology Co., Ltd., Shanghai, China; Jun Wei, wj@otcaix.iscas.ac.cn, Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences, Beijing, China.

ACM Reference Format:
Yuwei Zhang, Qingyuan Lu, Kai Liu, Wensheng Dou, Jiaxin Zhu, Li Qian, Chunxi Zhang, Zheng Lin, and Jun Wei. 2024. CITYWALK: Enhancing LLM-Based C++ Unit Test Generation via Project-Dependency Awareness and Language-Specific Knowledge. *ACM Trans. Softw. Eng. Methodol.* 37, 4, Article 111 (August 2024), 33 pages. https://doi.org/XXXXXXX.XXXXXXX

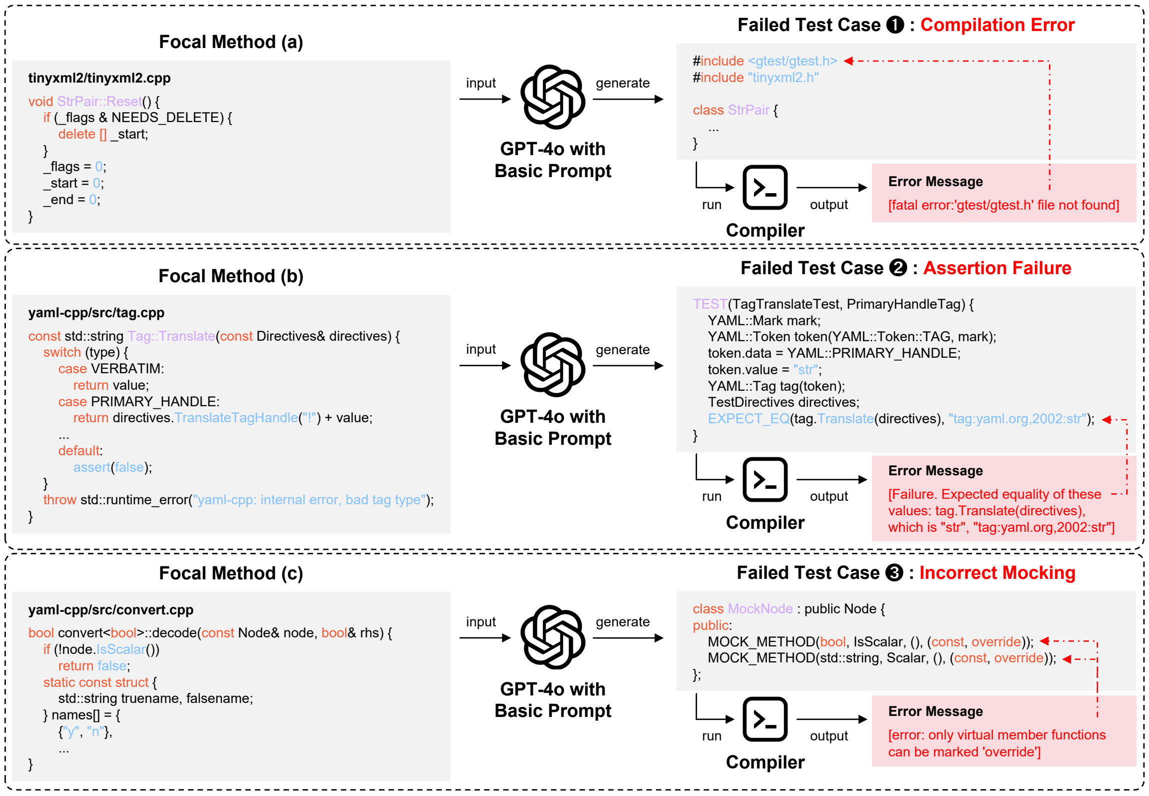
1 INTRODUCTION

The C++ programming language, widely renowned for its efficiency, scalability, and security, plays a crucial role in developing basic software such as operating systems and compilers. As these systems increase in size, however, ensuring code reliability becomes increasingly challenging in the fast-evolving software development process [23]. Unit testing [8, 29] serves as a fundamental technique in this pursuit, providing a robust means of validating individual software units in isolation from the rest of the system. By independently verifying each unit, developers can detect and rectify defects early in the software development lifecycle, thereby improving overall code quality. Nevertheless, writing high-quality C++ unit test cases becomes a challenging and time-consuming task for developers in practice when the complexity of software systems grows [14]. More recently, the application of large language models (LLMs) in unit test case generation has been extensively explored in both academia and industry [2, 3, 33, 41, 51]. For instance, Yuan et al. [51] conducted a comprehensive evaluation of using ChatGPT in automatically generating unit test cases for Java projects via both quantitative analysis and user studies. Their findings indicate that the ChatGPT-generated test cases exhibit commendable readability, thereby affirming the feasibility of an LLM-based technological approach.

Although LLM-based approaches have achieved remarkable performance in unit test case generation, the test code generated by LLMs still encounters issues, including compilation errors and assertion failures. Furthermore, current state-of-the-art approaches and tools [6, 30, 32, 51] primarily concentrate on interpreted programming languages such as Java and Python, with limited mature solutions yet available for generating unit test cases tailored to compiled programming language like C++. Particularly, the intricate language features of C++ present substantial challenges in utilizing LLMs to generate executable unit test cases with high coverage. To underscore the limitations of LLMs in the generation of C++ unit test cases, Figure 1 illustrates three motivating examples that demonstrate typical scenarios in which the most advanced LLM GPT-4o, when provided with the basic prompt[1], fails to generate correct unit test cases for the corresponding focal methods (i.e., the methods under test) in the open-source C++ projects.

- **Limitation 1: Missing code-agnostic contexts pertaining to the project configuration dependencies for LLMs to generate compilable code.** Existing LLM-based approaches [6, 51] leverage static analysis techniques to extract code dependency contexts related to the focal method from the focal class file, which aims to ensure the syntactic correctness of LLM-generated test code. However, LLMs may also produce compilation errors due to a lack of awareness regarding dependencies specified within the project's configuration. As shown in the **Error Message** output by **Failed Test Case ❶**, the test code generated by GPT-4o fails to compile because the configuration file for the `tinyxml2` project does not include the usage of the third-party library `gtest`. Given the absence of such code-agnostic contexts (e.g., the availability of specific programming framework) in the provided basic prompt, there is a high likelihood that GPT-4o will default to directly using the `GoogleTest` framework to generate C++ unit test cases for `tinyxml2`.
- **Limitation 2: Lack of cross-file intended behavior in the corresponding project as guidance for LLMs to generate correct assertions.** Considering the example of **Focal Method (b)**, the functionality of the `Translate`

[1]Following ChatTester [51], the basic prompt comprises the task description for unit test generation, a requirement to understand program's intent, the source code of the focal method, and the dependency contexts within the focal class file.



tinyxml2/tinyxml2.cpp

```
void StrPair::Reset() {
   if (_flags & NEEDS_DELETE) {
      delete [] _start;
   }
   _flags = 0;
   _start = 0;
   _end = 0;
}
```

```
#include <gtest/gtest.h>
#include "tinyxml2.h"

class StrPair {
   ...
}
```

```
[fatal error:'gtest/gtest.h' file not found]
```

yaml-cpp/src/tag.cpp

```
const std::string Tag::Translate(const Directives& directives) {
   switch (type) {
      case VERBATIM:
         return value;
      case PRIMARY_HANDLE:
         return directives.TranslateTagHandle("!") + value;
      ...
      default:
         assert(false);
   }
   throw std::runtime_error("yaml-cpp: internal error, bad tag type");
}
```

```
TEST(TagTranslateTest, PrimaryHandleTag) {
   YAML::Mark mark;
   YAML::Token token(YAML::Token::TAG, mark);
   token.data = YAML::PRIMARY_HANDLE;
   token.value = "str";
   YAML::Tag tag(token);
   TestDirectives directives;
   EXPECT_EQ(tag.Translate(directives), "tag:yaml.org,2002:str");
}
```

```
[Failure. Expected equality of these
values: tag.Translate(directives),
which is "str", "tag:yaml.org,2002:str"]
```

yaml-cpp/src/convert.cpp

```
bool convert<bool>::decode(const Node& node, bool& rhs) {
   if (!node.IsScalar())
      return false;
   static const struct {
      std::string truename, falsename;
   } names[] = {
      {"y", "n"},
      ...
}
```

```
class MockNode : public Node {
public:
   MOCK_METHOD(bool, IsScalar, (), (const, override));
   MOCK_METHOD(std::string, Scalar, (), (const, override));
};
```

```
[error: only virtual member functions
can be marked 'override']
```

Fig. 1. Limitations of LLMs in Generating C++ Unit Test Cases.

method in the `tag` class file is to invoke the relevant interface methods provided by the `Directives` class to translate various YAML tags in accordance with their respective types. In this instance, the variable `type` is defined within the header file of `tag`, while the contextual information related to the invoked interface methods resides in the class file of `Directives`. Given that these cross-file dependency contexts within the `yaml-cpp` project are not encompassed in the basic prompt, comprehending the functional intent of `Translate` alone does not sufficiently assist GPT-4o in grasping the usage of `Translate`. Consequently, **Failed Test Case ❷** is unable to configure the correct YAML tag type to align with the target translation pattern. This misalignment causes the actual outcome to deviate from the expected result, consequently triggering an assertion failure.

- **Limitation 3: LLMs struggle to correctly generate complex test code without an understanding of C++ language-specific domain knowledge.** In practice, the focal method may need to interact with complex data objects defined in dependent class files. In such cases, developers typically create virtual objects using mocking techniques [31, 42] to simulate the behavior of real external dependencies. However, to effectively generate complex test code, LLMs must grasp the principles and applicable scenarios related to mocking [38, 39]. In **Focal Method (c)**, `decode` needs to return a boolean value by verifying whether `node` is a scalar. As illustrated in **Failed Test Case ❸**, GPT-4o attempts to simulate a `Node` class object to mimic the test input. The **Error Message** indicates that the methods invoked on `node` are non-virtual functions. As a consequence, GPT-4o generates test code that contains compilation errors, stemming from an insufficient understanding of such complex language feature of C++ mocking.

To tackle the aforementioned limitations, we propose CITYWALK, a novel framework designed to enhance the capabilities of LLMs in generating high-quality **C**++ un**I**t **T**est cases by providing project-dependenc**Y** a**WA**reness and **L**anguage-specific **K**nowledge. Our key insight lies in *enabling LLMs to act as skilled developers, leveraging specialized knowledge and global comprehension of project dependencies to perform unit testing effectively.* The main ideas of CITYWALK are outlined as follows:

- **Empowering LLMs with an awareness of project-level dependency relationships through program analysis.** The project dependencies employed by CITYWALK are categorized into two types: cross-file data dependencies and configuration dependencies. To conduct cross-file data dependency analysis, CITYWALK initially performs static analysis to identify the header and source code files within the project that exhibit a dependency-chain relationship with the focal class file. Subsequently, CITYWALK utilizes the abstract syntax tree (AST) to extract the relevant cross-file data dependency contexts associated with the focal method from these identified files. Furthermore, our observations indicate that when the project under test necessitates specific versions of the compilation environment or third-party libraries, the probability of encountering compilation errors in the LLM-generated unit tests increases substantially. By analyzing project's configuration files, CITYWALK gathers critical information regarding the third-party library usage and the requirements of the compilation environment. These configuration dependencies are explicitly provided to LLMs to reduce the occurrence of compilation errors.
- **Augmenting LLMs with intention-guiding information related to the focal method via a hybrid retrieval strategy.** Software repositories often contain extensive documentation, such as requirements specifications and API references [19]. Utilizing this natural language information can further aid LLMs in understanding the functional logic of the focal method. Moreover, when the focal method has long dependency chains or complex initialization of dependent objects, incorporating relevant code snippets, especially those that invoke the focal method or demonstrate its initialization, into the prompts can greatly help guide LLMs in understanding the real-world usage patterns of the focal method, thereby compensating for the limitations of static analysis techniques. To facilitate the retrieval of both natural language documentation and programming language code snippets, CITYWALK employs a hybrid strategy designed to efficiently extract intention-guiding information from bimodal sources, ensuring an accurate and contextually relevant retrieval process.
- **Prompting LLMs with step-by-step instructions and language-specific knowledge derived from empirical observations.** CITYWALK decomposes the unit test generation task into smaller, more specific subtasks with structured step-by-step instructions. The goal is to streamline the generation of C++ unit test cases by providing LLMs with detailed procedural guidance. Within CITYWALK, LLMs are utilized to: (1) infer both the intention and dependencies of the focal method; (2) generate an entire test file for the focal method by leveraging the provided guidance; and (3) refine the generated test cases using language-specific knowledge. To perform Step (3), we further conduct an empirical analysis of the LLM-generated failed test cases for building language-specific domain knowledge tailored to C++. This knowledge-driven approach assists in guiding LLMs to emulate experienced developers in generating accurate test cases.

We implement a prototype of CITYWALK using GPT-4o and evaluate it on a collected benchmark comprising 1288 focal methods from ten C++ projects. We conduct a comparative analysis of CITYWALK against seven state-of-the-art baselines. Our evaluation demonstrates that CITYWALK outperforms all baselines in both compilation success rate and code coverage. Our contributions can be summarized as follows:

- We present the first attempt at enhancing the capabilities of LLMs for C++ unit test generation by leveraging project-dependency awareness and language-specific knowledge, enabling LLMs to function like human developers in writing correct unit test cases.
- We thoroughly evaluate CITYWALK on a diverse collection of open-source projects, thereby substantiating the effectiveness of each component within CITYWALK. Furthermore, we illustrate its capacity to generalize in generating high-quality unit tests across various LLMs.
- We publicly release the artifacts [53] of CITYWALK on Zenodo to facilitate the reproduction.

Article Organization. The remainder of this paper is organized as follows: Section 2 introduces in detail the proposed framework. Section 3 and Section 4 provide the experimental setups and results of our research. Section 5 presents multi-perspective discussions and discloses the threats to the validity of our approach. Section 6 describes the related work. Section 7 concludes with research findings and future directions.

2 METHODOLOGY

In this paper, we introduce CITYWALK, a novel LLM-based framework for C++ unit test generation that addresses the limitations outlined in Section 1. As illustrated in Figure 2, CITYWALK consists of five stages designed to enhance LLMs in generating high-quality C++ unit test cases. It effectively integrates the strengths of program analysis with retrieval-augmented strategies, empowering LLMs with developer-like capabilities through three key aspects of guidance.

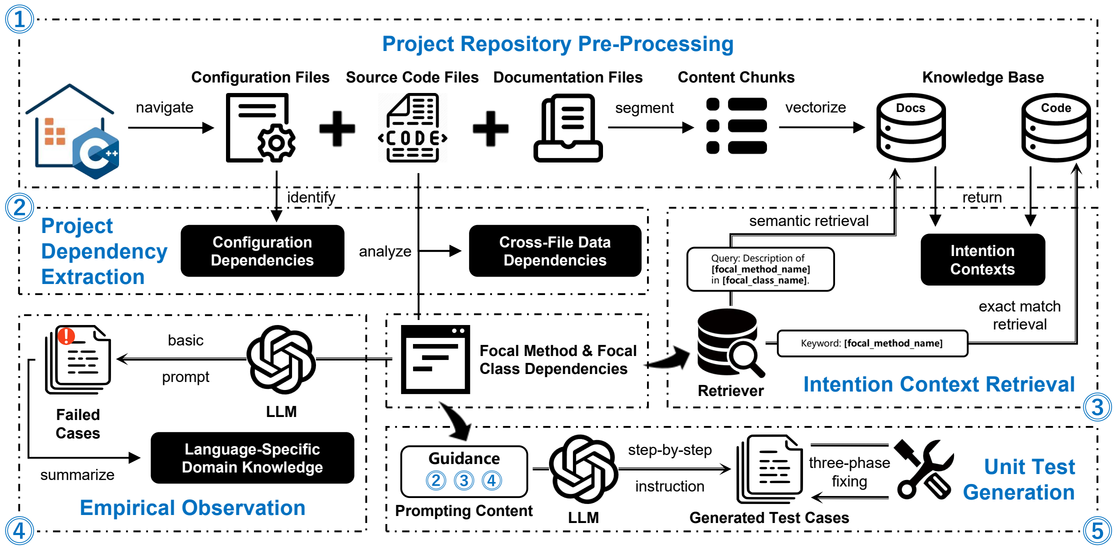


Fig. 2. Overview of CITYWALK.

2.1 Task Formulation

We first provide a formal definition of the unit test generation task. Let $\mathtt{Repo} = \{\mathtt{SC}, \mathtt{C}, \mathtt{D}\}$ be the project repository, which includes a set of source code files $\mathtt{SC} = \{\mathtt{sc}_1, \mathtt{sc}_2, ...\mathtt{sc}_i\}$, configuration files $\mathtt{C} = \{\mathtt{c}_1, \mathtt{c}_2, ...\mathtt{c}_j\}$, and documentation files

$\mathtt{D} = \{\mathtt{d}_1, \mathtt{d}_2, \ldots \mathtt{d}_k\}$. The methods within Repo are denoted as $\mathtt{M} = \bigcup_{\mathtt{sc} \in \mathtt{SC}} \mathtt{M}(\mathtt{sc})$, where $\mathtt{M}(\mathtt{sc}) = \{\mathtt{m}_1, \mathtt{m}_2, \ldots, \mathtt{m}_n\}$ is the set of methods in sc. Given a focal method $\mathtt{m}_{\mathtt{focal}} \in \mathtt{M}(\mathtt{sc}_{\mathtt{focal}})$ associated with the focal class file $\mathtt{sc}_{\mathtt{focal}}$, the objective of CITYWALK is to generate unit test case(s) TC for $\mathtt{m}_{\mathtt{focal}}$ through the following stages:

① **Project Repository Pre-Processing:** This stage involves navigating through all files in Repo, segmenting the selected files into chunks, and storing the vectorized chunks in the knowledge bases $\mathtt{KB}_{\mathtt{code}}$ and $\mathtt{KB}_{\mathtt{docs}}$.

② **Project Dependency Extraction:** In this stage, we identify configuration dependencies within C and cross-file data dependencies between $\mathtt{sc}_{\mathtt{focal}}$ and SC via program analysis.

③ **Intention Context Retrieval:** This stage focuses on retrieving relevant code snippets from $\mathtt{KB}_{\mathtt{code}}$ and documentation from $\mathtt{KB}_{\mathtt{docs}}$ as intention contexts utilizing a hybrid strategy.

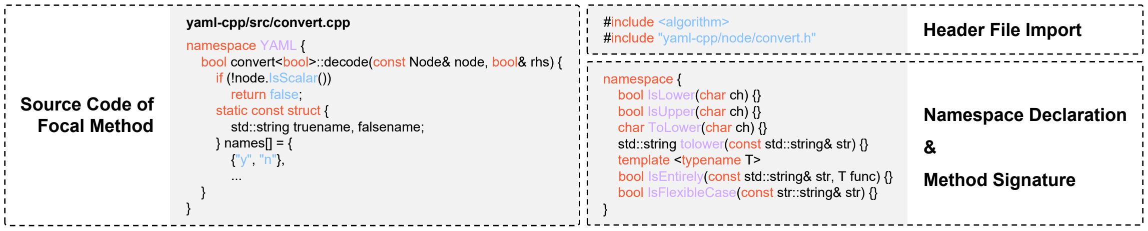
④ **Empirical Observation:** In this stage, we analyze the LLM-generated failed test cases and summarize the language-specific knowledge derived from empirical observations.

⑤ **Unit Test Generation:** This stage involves prompting the LLM with step-by-step instructions to generate test cases $\mathtt{TC} \longleftarrow \mathtt{LLM}(\mathtt{PROMPT})$, where PROMPT includes the basic prompt supplemented with additional guidance from **Stage** ②-④, and incorporates a three-phase post-processing approach for error-fixing.

2.2 Project Repository Pre-Processing

In the context of program analysis-based and retrieval-augmented unit test generation, this stage involves two preliminary processes for pre-processing the given focal method and corresponding project. These processes are essential to help LLMs understand the code structure and its dependencies, thereby facilitating precise analysis and effective retrieval to support test case generation.

2.2.1 Structured Focal Context Extraction. Given the input focal class file, CITYWALK begins by parsing the source code into an AST using Clang[2]. For each focal method, CITYWALK then gathers the essential focal contexts in a structured format, providing the foundational information necessary for project dependency analysis. Figure 3 illustrates the structured focal contexts for decode in the `convert` class, preserving the complete code implementation of the focal method, along with imports from C++ standard libraries, third-party libraries, user-defined header files, namespace declarations, and signatures of other methods within the focal class.


Source Code of Focal Method
Header File Import
Namespace Declaration & Method Signature

yaml-cpp/src/convert.cpp

```
namespace YAML {
   bool convert<bool>::decode(const Node& node, bool& rhs) {
      if (!node.IsScalar())
         return false;
      static const struct {
         std::string truename, falsename;
      } names[] = {
         {"y", "n"},
         ...
   }
}
```

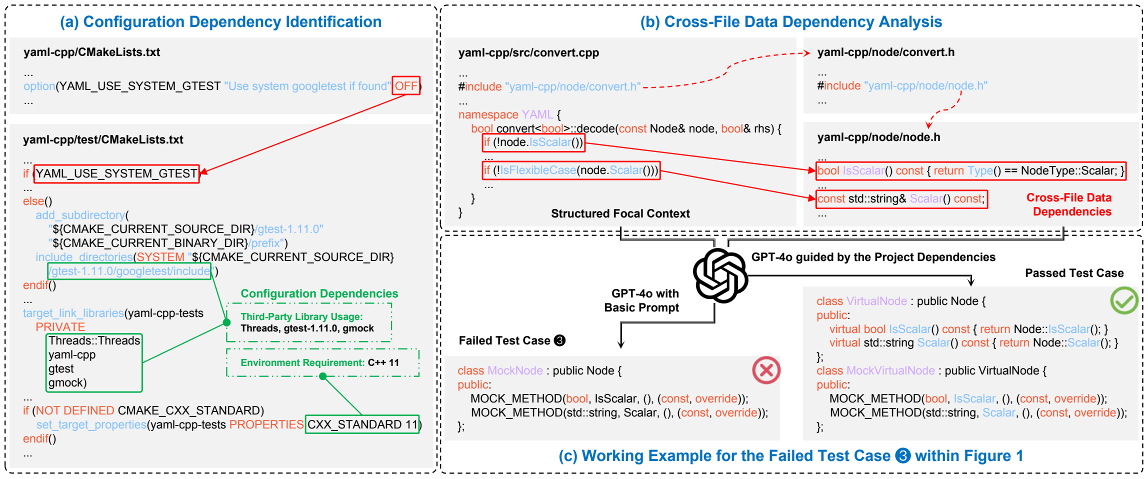
```
#include <algorithm>
#include "yaml-cpp/node/convert.h"
```

```
namespace {
   bool IsLower(char ch) {}
   bool IsUpper(char ch) {}
   char ToLower(char ch) {}
   std::string tolower(const std::string& str) {}
   template <typename T>
   bool IsEntirely(const std::string& str, T func) {}
   bool IsFlexibleCase(const str::string& str) {}
}
```

Fig. 3. Illustrative Example of Structured Focal Context for the decode Method in the `convert` Class.

2.2.2 Knowledge Base Construction. Retrieval-augmented generation (RAG) allows LLMs to leverage information from external knowledge bases for reducing hallucinations [5]. To enhance the LLM's understanding of the focal method during unit test generation, CITYWALK constructs knowledge bases using the documentation and source code from

[2]https://clang.llvm.org


(a) Configuration Dependency Identification
yaml-cpp/CMakeLists.txt
yaml-cpp/test/CMakeLists.txt
Configuration Dependencies
Third-Party Library Usage: Threads, gtest-1.11.0, gmock
Environment Requirement: C++ 11
(b) Cross-File Data Dependency Analysis
yaml-cpp/src/convert.cpp
yaml-cpp/node/convert.h
yaml-cpp/node/node.h
Structured Focal Context
Cross-File Data Dependencies
GPT-4o guided by the Project Dependencies
GPT-4o with Basic Prompt
Failed Test Case ❸
Passed Test Case
(c) Working Example for the Failed Test Case ❸ within Figure 1

```
...
option(YAML_USE_SYSTEM_GTEST "Use system googletest if found" OFF)
...
```

```
...
if (YAML_USE_SYSTEM_GTEST)
...
else()
  add_subdirectory(
     "${CMAKE_CURRENT_SOURCE_DIR}/gtest-1.11.0"
     "${CMAKE_CURRENT_BINARY_DIR}/prefix")
  include_directories(SYSTEM "${CMAKE_CURRENT_SOURCE_DIR}
     /gtest-1.11.0/googletest/include")
endif()
...
target_link_libraries(yaml-cpp-tests
  PRIVATE
     Threads::Threads
     yaml-cpp
     gtest
     gmock)
...
if (NOT DEFINED CMAKE_CXX_STANDARD)
  set_target_properties(yaml-cpp-tests PROPERTIES CXX_STANDARD 11)
endif()
...
```

```
...
#include "yaml-cpp/node/convert.h"
...
namespace YAML {
  bool convert<bool>::decode(const Node& node, bool& rhs) {
     if (!node.IsScalar())
     ...
     if (!IsFlexibleCase(node.Scalar()))
     ...
  }
}
```

```
...
#include "yaml-cpp/node/node.h"
...
```

```
...
bool IsScalar() const { return Type() == NodeType::Scalar; }
...
const std::string& Scalar() const;
...
```

```
class MockNode : public Node {
public:
  MOCK_METHOD(bool, IsScalar, (), (const, override));
  MOCK_METHOD(std::string, Scalar, (), (const, override));
};
```

```
class VirtualNode : public Node {
public:
  virtual bool IsScalar() const { return Node::IsScalar(); }
  virtual std::string Scalar() const { return Node::Scalar(); }
};
class MockVirtualNode : public VirtualNode {
public:
  MOCK_METHOD(bool, IsScalar, (), (const, override));
  MOCK_METHOD(std::string, Scalar, (), (const, override));
};
```

Fig. 4. Illustrative Example of Project Dependency Extraction.

the project repository. First, CITYWALK scans all semantically relevant files within the repository, including project documentation (e.g., requirement specifications) written in natural language and source code files in programming languages. Files that do not contribute semantic knowledge (e.g., configuration files) are excluded from constructing knowledge bases. CITYWALK then applies text segmentation strategies to slice the documentation and code files. For instance, markdown files can be segmented into text chunks based on title levels, while source code files are divided into method-level code chunks. Additionally, natural language comments within the code are extracted separately as text chunks. To facilitate efficient semantic retrieval and similarity calculations, CITYWALK utilizes an embedding model BGE [46] for the vectorization of segmented text chunks, storing the resulting vectors in $\mathsf{KB}_{\mathsf{docs}}$. The segmented code chunks are directly stored in $\mathsf{KB}_{\mathsf{code}}$ for exact match retrieval without vectorization.

2.3 Project Dependency Extraction

LLM-generated test code often fails to compile, typically due to missing dependencies or specific version requirements of the compilation environment. This limitation compels LLMs to make assumptions about the usage of undefined variables or methods based on their reasoning [55]. To mitigate compilation errors, CITYWALK enables LLMs to be aware of relevant dependency contexts at the project level through program analysis. The project dependencies extracted by CITYWALK are classified into the following two categories:

- **Configuration Dependencies:** As illustrated in Figure 4(a), CITYWALK identifies both the **usage of third-party libraries** and the **compilation environment requirements** of the project under test. CITYWALK first parses the `CMakeLists` files in the `root` and `test` dictionaries using regular expressions to extract configuration dependencies based on keyword-matching, including the command like `target_link_libraries`. This process captures details about dependent third-party libraries and their versions (e.g., **gtest-1.11.0**) as well as the compilation environment requirement (i.e., **C++ 11**) specified through `set_target_properties`. The parsed dependencies (framed by the green rectangle) are stored in a structured format that facilitates seamless integration with LLM prompts. This enables LLMs to understand the necessary code-agnostic dependencies of the project under test, which is essential for generating compilable test code.

- **Cross-File Data Dependencies:** In addition to the focal contexts utilized by existing LLM-based approaches [6, 51], CITYWALK conducts data dependency analysis via AST to gather cross-file dependencies as contextual guidance, helping LLMs in reasoning accurately with sufficient information. CITYWALK first navigates all files with specific suffixes (i.e., ".cpp" and ".h"). In C++, user-defined header files are typically imported using `#include ""`. Thus, CITYWALK utilizes program analysis to filter out files with direct dependencies on the focal class file based on the `include` field information. This analysis process is recursive, and CITYWALK limits the extracted dependency chain to two layers. This design is primarily motivated by the context window constraints of employed LLMs, ensuring that the length of extracted contextual dependencies remains within the model's processing capacity. Subsequently, CITYWALK employs data-flow analysis on the AST nodes of the invoked methods within the focal method to extract cross-file data dependency contexts from the filtered files. As shown in Figure 4(b), CITYWALK extracts the dependent contexts (framed by the red rectangle) for the invoked methods (i.e., `IsScalar` and `Scalar`) in `decode` from the `Node` class through a two-layer relationship (i.e., `convert.cpp` ⇢ `convert.h` ⇢ `node.h`). Furthermore, Figure 4(c) presents a working example of how cross-file data dependencies can guide LLMs in generating executable test code for **Failed Test Case ❸** shown in Figure 1. By incorporating additional contextual information from `Node` and prompting LLMs with the language-specific knowledge of `gmock`, GPT-4o derives a class and declares virtual functions, successfully simulating the behavior of the invoked methods within `decode` and generating accurate test inputs.

2.4 Intention Context Retrieval

To better guide LLMs in understanding the program intention of the focal method, CITYWALK employs RAG to generate effective unit test cases by leveraging the project documentation and source code stored in the knowledge bases described in Section 2.2.2. The natural language documentation provides functionality requirements of the focal method as articulated by developers, while the source code snippets within the project offer LLMs real-world examples of focal method invocation and initialization. This approach mitigates issues related to focal methods that have dependency chains longer than two layers or are complex to initialize. Specifically, CITYWALK employs a hybrid retrieval strategy that conducts knowledge queries in two stages. For $\mathtt{KB}_{\mathtt{docs}}$, the process begins by generating a corresponding query statement (as shown in Figure 2) based on the name of the focal method and the class to which it belongs. The embedding model BGE is then utilized to convert this query into a vector representation, capturing its semantic information and enabling alignment with the content of $\mathtt{KB}_{\mathtt{docs}}$ in vector space. Using the retrieval algorithms provided by the vector database Faiss [11], the similarity between the query vector and the vectors in $\mathtt{KB}_{\mathtt{docs}}$ is computed, employing cosine similarity to identify the vectors that are most similar to the query vector. Based on the similarity scores, the top-2 most relevant responses are selected as retrieval results. This setup ensures retrieval accuracy while providing sufficiently detailed guidance information, assisting in the generation of more comprehensive test cases. Preliminary practices indicate that semantic retrieval for code snippet examples often yields imprecise results [4, 48]. To optimize retrieval from $\mathtt{KB}_{\mathtt{code}}$, CITYWALK adopts an exact match strategy based on the focal method's signature, including its name and parameter types, extracted through static analysis. Specifically, CITYWALK first utilizes the regular expression: `r‘\b’` + `re.escape(focal_method_name)` + `r‘\s*\([^)]*\)’` to identify candidate method invocation or initialization examples from $\mathtt{KB}_{\mathtt{code}}$. Following this, a verification step filters out candidate examples with mismatched parameter counts or types, ensuring only those that exactly match the focal method's signature are retained. For example, `focal_method_name(double)` will not match `focal_method_name(int)`. This two-step strategy guarantees accurate retrieval while effectively distinguishing between similar method variants.

2.5 Empirical Observation

In addition to the extracted project dependencies and retrieved intention contexts, CITYWALK further provides error patterns, along with their corresponding solution guidelines, as language-specific domain knowledge. This knowledge tailored to C++ programming language is used to prompt LLMs with instructions aimed at mitigating common errors in the task of unit test generation. To achieve this, we conduct an empirical study to evaluate the correctness of C++ test cases generated by existing state-of-the-art LLMs through quantitative analysis. The procedure for our empirical investigation is outlined as follows:

- **Empirical Setups:** (1) **Benchmark.** We adopt a diverse set of real-world open-source C++ projects collected from GitHub, with the selection criteria and detailed statistics provided in Section 3.2. (2) **Subject LLMs.** We select two state-of-the-art LLMs to evaluate their capabilities in generating high-quality C++ unit test cases based on our collected benchmark: the open-source LLM DeepSeek-V3 [10] and the closed-source LLM GPT-4o [22]. (3) **Prompt Design.** As suggested by Yuan et al. [51], we design our prompt by closely following common practices in recent unit test generation research. The prompt consists of a natural language description that explains the task to the LLM, along with the code context, which includes the focal method and other relevant contextual information (e.g., the fields and method signatures within the focal class).
- **Experimental Procedure:** For each project in the benchmark, we clone its repository from GitHub and extract relevant information to support prompt construction. We query the selected LLMs using our designed prompt for each focal method and consider the test cases generated by the LLMs as the output. We use the official API of each LLM with the configuration set to generate the top-1 chat completion choice and a sampling temperature of 0. The generated test cases are then placed in the test directory of the project, where we attempt to compile and execute them for subsequent analysis. In our experimental design, we evaluate the correctness of the generated test cases from three perspectives. First, we use the Clang parser as a syntax checker to verify the syntactic correctness of the generated test cases. Next, we measure the correctness of compilation and execution by verifying whether the generated test cases compile successfully and run without errors. Error messages produced during compilation and execution are automatically extracted for analysis. To examine the failed test cases generated by the two evaluated LLMs, the first two authors manually classified the errors based on the corresponding compiler-generated messages. This empirical process involved over 1000 failed test cases and required approximately five hours. To ensure consistency and accuracy, the two authors collaboratively reviewed and resolved discrepancies through discussion, reaching consensus on all classification results. In addition, we employ `llvm-cov`[3] to collect both line and branch coverage, providing an assessment of the sufficiency of the generated test cases.
- **Error Analysis:** To better understand the limitations of existing LLMs in C++ unit test generation, we analyze common error patterns in the failed test cases generated by DeepSeek-V3 and GPT-4o, with a particular focus on their impact on compilation correctness. It is important to note that this analysis is limited to **compilation errors**, which constitute a significant proportion of all errors and represent a critical initial barrier to the successful execution and correctness of generated test cases. Specifically, we automatically categorize each failed test case based on the associated compilation error message. Table 1 presents the distribution of compilation errors observed in the test cases generated by the evaluated LLMs. The “Frequency” column reports the number of occurrences for each error pattern across **DeepSeek-V3**, **GPT-4o**, and their combined total. For clarity, only

[3] https://llvm.org/docs/CommandGuide/llvm-cov.html

Table 1. Compilation Error Breakdown for LLM-Generated Test Cases

Error Pattern	**Error Description**	**Frequency**		
		DeepSeek-V3	**GPT-4o**	**Total**
Undefined Symbols Error	Missing or unresolved identifiers	351	382	733
Access Error	Invalid access to class members	157	147	304
Type Error	Type mismatches in expressions or assignments	140	161	301
Other	Miscellaneous unclassified errors	91	76	167
Test Setup Error	Failure during initialization of tests	51	99	150
Linker Error	Cross-file linkage failure	92	32	124
Syntax Error	Invalid syntax in C++ source files	18	59	77
Namespace Error	Incorrect or missing namespace usage	0	45	45
Multiple Definition Error	Duplicate symbols defined in different files	5	26	31
Template Error	Invalid usage of C++ templates	1	9	10

Table 2. Language-Specific Domain Knowledge Derived from Empirical Observations

Category	**Guideline**
(A) Compilation Error	(A.1) Import all necessary dependencies with the correct paths. (A.2) Use only the C++ standard libraries, imported third-party libraries, and provided methods. (A.3) If `gtest` is not allowed, directly call test methods from the main function. (A.4) Use the correct namespace throughout the tests. (A.5) Properly handle static members by accessing them using the class name. (A.6) Avoid invoking private methods or accessing private fields defined in the program.
(B) Execution Failure	(B.1) Choose appropriate assertions for the pointer data type, clearly distinguishing between address and content comparisons. (B.2) For mocking (if using `gmock`), remember that only virtual methods can be mocked.
(C) Poor Coverage	(C.1) Ensure coverage of true and false branches for each conditional predicate at least once. (C.2) Utilize non-terminating assertions (e.g., `EXPECT_*`) to maximize code coverage.

error patterns that appeared more than ten times are included in the table. As shown in Table 1, LLM-generated test cases exhibit a diverse range of compilation error patterns. The most common errors are related to undefined symbols, typically caused by references to unresolved identifiers such as undeclared methods or variables. Two other prevalent error patterns include access violations and type mismatches. Access errors generally result from invalid attempts to reference class members, while type errors stem from incompatible expressions or incorrect type assignments. Additionally, we observe that **GPT-4o** frequently produces test code with incorrect namespace usage, which is not present in **DeepSeek-V3** outputs. This quantitative analysis reveals persistent patterns of compilation errors encountered by LLMs during unit test generation and provides valuable insights that inform the design of our mitigation strategies.

- **Solution Guidelines:** As presented in the first column of Table 2, we manually group similar errors into three high-level categories: **(A) Compilation Errors**, **(B) Execution Failures**, and **(C) Poor Coverage**. While compilation errors constitute the majority, the LLM-generated test cases also exhibit execution failures, often caused by incorrect assertions or flawed mocking code. Furthermore, poor coverage emerges as a significant issue, largely attributable to the LLMs' limited ability to exercise all conditional branches and their misuse of assertion types. As listed in the second column of Table 2, we systematically summarize solution guidelines for the corresponding error patterns based on empirical observations of the failed test cases generated by the selected LLMs. As discussed in Section 2.3, **Failed Test Case ❸** can be addressed by leveraging additional cross-file data dependencies along with **Guideline (B.2)**. Specifically, we organize **Guideline (B.2)** as instructions to prompt LLMs in a knowledge-driven manner for tackling the incorrect mocking issue. In doing so, CITYWALK incorporates language-specific knowledge about mocking in C++, including best practices for creating mock objects for virtual functions. In the case of the `decode` method, CITYWALK provides LLMs with detailed guidance on how to correctly mock non-virtual member functions in C++. By enhancing the LLM's understanding of these language-specific nuances, CITYWALK ensures the generated mocking code is both syntactically and semantically correct, effectively preventing execution failures caused by improper mocking of non-virtual methods. While these guidelines do not cover all error types, they are instrumental in guiding LLMs to produce high-quality test cases and mitigate common errors in C++ unit test generation.

2.6 Unit Test Generation

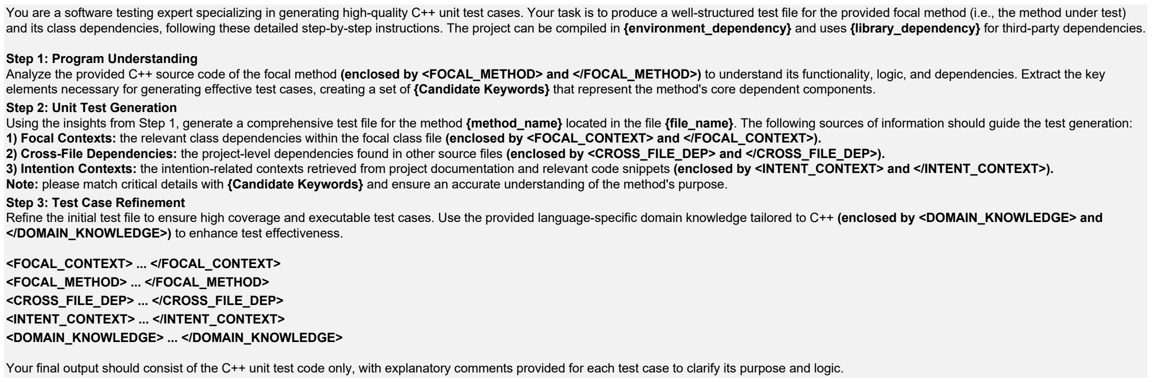
Unit test generation is a complex task that poses significant challenges when generating high-quality test cases from scratch with limited guidance. To address this, CITYWALK initially produces an entire test file for the given focal method by following step-by-step instructions. Subsequently, CITYWALK corrects the failed test cases using a three-phase rule-based fixing approach. As illustrated in Algorithm 1, we provide detailed process overview as follows.

Algorithm 1: Unit Test Case Generation for a Focal Method.

```
Input: The given focal method: $\mathtt{m}_{\mathtt{focal}}$; The focal contexts: $\mathtt{Context}_{\mathtt{focal}}$; The configuration dependencies: $\mathtt{Dep}_{\mathtt{c}}$; The cross-file
       data dependencies: $\mathtt{Dep}_{\mathtt{d}}$; The intention contexts: $\mathtt{Context}_{\mathtt{intent}}$; The guidelines of language-specific domain
       knowledge: $\mathtt{Guideline}_{\mathtt{DK}}$; The step-by-step instruction prompt: $\mathtt{PROMPT}_{\mathtt{step}}$; The syntactic error fixing rules: $\mathtt{Rule}_{\mathtt{F_s}}$;
       The compilation error fixing rules: $\mathtt{Rule}_{\mathtt{F_c}}$; The error fixing prompt: $\mathtt{PROMPT}_{\mathtt{F}}$
Output: The generated unit test case(s): TC
1  /* Initial Unit Test Case Generation via Step-by-Step Instructions */
2  /* Step 1: Program Understanding */
3  $\mathtt{Intent}, \mathtt{Dep}_{\mathtt{ingredient}} \leftarrow \mathbf{LLM}(\mathtt{PROMPT}_{\mathtt{step}}(\mathtt{m}_{\mathtt{focal}}))$
4  /* Step 2: Unit Test Generation */
5  $\mathtt{TC} \leftarrow \mathbf{LLM}(\mathtt{PROMPT}_{\mathtt{step}}(\mathtt{m}_{\mathtt{focal}}, \mathtt{Context}_{\mathtt{focal}}, \mathtt{Dep}_{\mathtt{c}}, \mathtt{Dep}_{\mathtt{d}}, \mathtt{Context}_{\mathtt{intent}}, \mathtt{Intent}, \mathtt{Dep}_{\mathtt{ingredient}}))$
6  /* Step 3: Test Case Refinement */
7  $\mathtt{TC} \leftarrow \mathbf{LLM}(\mathtt{PROMPT}_{\mathtt{step}}(\mathtt{TC}, \mathtt{Guideline}_{\mathtt{DK}}))$
8  /* Post-Processing of the LLM-Generated Unit Test Cases */
9  $\mathtt{TC} \leftarrow \mathtt{Rule}_{\mathtt{F_s}}(\mathtt{TC})$
10 if Compiler(TC) is not PASS then
11     $\mathtt{TC} \leftarrow \mathtt{Rule}_{\mathtt{F_c}}(\mathtt{TC})$
12     if Compiler(TC) is not PASS then
13         $\mathtt{Context}_{\mathtt{error}} \leftarrow \mathbf{Compiler}(\mathtt{TC})$
14         $\mathtt{TC} \leftarrow \mathbf{LLM}(\mathtt{PROMPT}_{\mathtt{F}}(\mathtt{TC}, \mathtt{Context}_{\mathtt{error}}))$
15 return TC
```

2.6.1 *Initial Unit Test Case Generation via Step-by-Step Instructions.*

As shown in Figure 5, the prompting content for querying the LLM comprises four components: the **task definition**, the **step-by-step instructions**, the **contextual information**, and the **output format**. Within the context of CITYWALK, the LLM is employed to generate initial test cases by following steps:



You are a software testing expert specializing in generating high-quality C++ unit test cases. Your task is to produce a well-structured test file for the provided focal method (i.e., the method under test) and its class dependencies, following these detailed step-by-step instructions. The project can be compiled in **{environment_dependency}** and uses **{library_dependency}** for third-party dependencies.

Step 1: Program Understanding
Analyze the provided C++ source code of the focal method **(enclosed by <FOCAL_METHOD> and </FOCAL_METHOD>)** to understand its functionality, logic, and dependencies. Extract the key elements necessary for generating effective test cases, creating a set of **{Candidate Keywords}** that represent the method's core dependent components.
Step 2: Unit Test Generation
Using the insights from Step 1, generate a comprehensive test file for the method **{method_name}** located in the file **{file_name}**. The following sources of information should guide the test generation:
1) Focal Contexts: the relevant class dependencies within the focal class file **(enclosed by <FOCAL_CONTEXT> and </FOCAL_CONTEXT>).**
2) Cross-File Dependencies: the project-level dependencies found in other source files **(enclosed by <CROSS_FILE_DEP> and </CROSS_FILE_DEP>).**
3) Intention Contexts: the intention-related contexts retrieved from project documentation and relevant code snippets **(enclosed by <INTENT_CONTEXT> and </INTENT_CONTEXT>).**
Note: please match critical details with **{Candidate Keywords}** and ensure an accurate understanding of the method's purpose.
Step 3: Test Case Refinement
Refine the initial test file to ensure high coverage and executable test cases. Use the provided language-specific domain knowledge tailored to C++ **(enclosed by <DOMAIN_KNOWLEDGE> and </DOMAIN_KNOWLEDGE>)** to enhance test effectiveness.

<FOCAL_CONTEXT> ... </FOCAL_CONTEXT>
<FOCAL_METHOD> ... </FOCAL_METHOD>
<CROSS_FILE_DEP> ... </CROSS_FILE_DEP>
<INTENT_CONTEXT> ... </INTENT_CONTEXT>
<DOMAIN_KNOWLEDGE> ... </DOMAIN_KNOWLEDGE>

Your final output should consist of the C++ unit test code only, with explanatory comments provided for each test case to clarify its purpose and logic.

Fig. 5. The Detailed Prompting Content for Generating Initial Test Cases for a Given Focal Method.

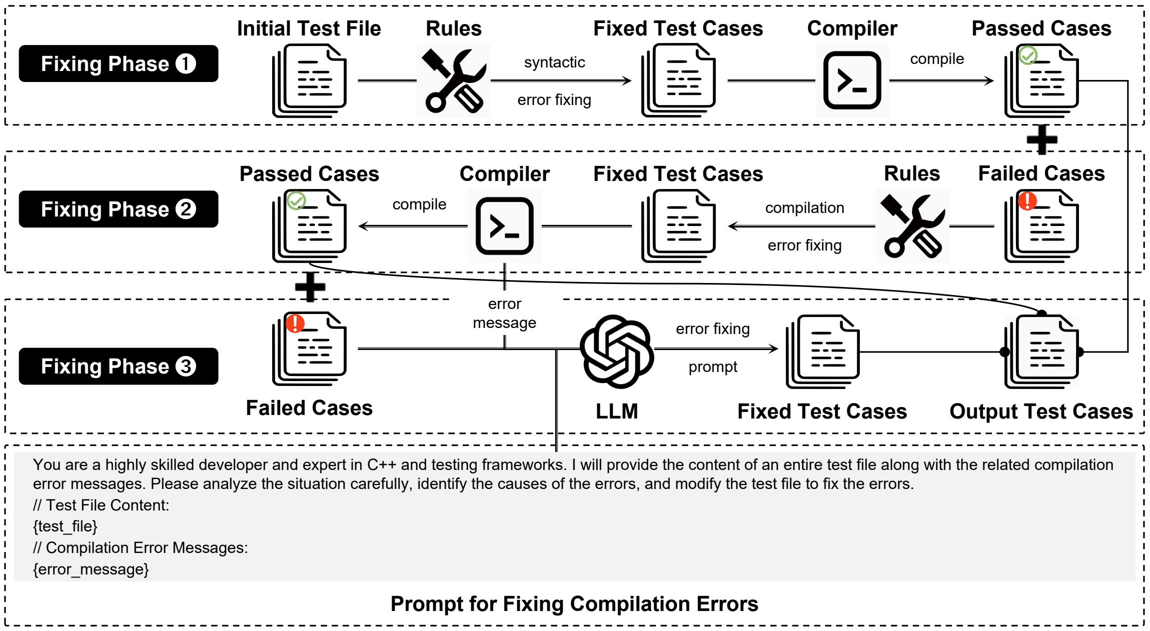
(1) *Program Understanding* **(Line 3)**: First, the LLM analyzes the provided source code of $\mathtt{m}_{\mathtt{focal}}$ to grasp its intended functionality, denoted as `Intent`. Next, the LLM extracts the key elements necessary for generating effective test cases, creating a set of candidate keywords (i.e., $\mathtt{Dep}_{\mathtt{ingredient}}$) that represent the core dependent ingredients of $\mathtt{m}_{\mathtt{focal}}$.
(2) *Unit Test Generation* **(Line 5)**: Based on the `Intent` of $\mathtt{m}_{\mathtt{focal}}$, the LLM generates an initial test file for $\mathtt{m}_{\mathtt{focal}}$ by utilizing the provided contextual information, which includes the focal contexts $\mathtt{Context}_{\mathtt{focal}}$, project dependencies $\mathtt{Dep}_{\mathtt{c}}$ and $\mathtt{Dep}_{\mathtt{d}}$, intention contexts $\mathtt{Context}_{\mathtt{intent}}$. Additionally, the LLM matches the contextual information with the extracted $\mathtt{Dep}_{\mathtt{ingredient}}$ to ensure an precise understanding of the purpose of $\mathtt{m}_{\mathtt{focal}}$.
(3) *Test Case Refinement* **(Line 7)**: To enhance the effectiveness of the LLM-generated test cases `TC`, the LLM refines the initial test file using the language-specific domain knowledge as guidelines $\mathtt{Guideline}_{\mathtt{DK}}$, ensuring the generation of high coverage and executable `TC`.

2.6.2 Post-Processing of the LLM-Generated Unit Test Cases. Inspired by ChatUniTest [6], CITYWALK applies post-processing techniques to fix `TC` that contain syntactic and compilation errors. As shown in Figure 6, the fixing process consists of the following three phases:

❶ **Rule-based Syntactic Error Fixing**: CITYWALK first utilizes $\mathtt{Rule}_{\mathtt{F_s}}$ to address the syntactic errors in `TC` **(Line 9)**. The syntactic error fixing rules include:
- Ensuring all brackets are closed through pattern matching.
- Removing all incorrect import statements by consolidating imports within $\mathtt{Context}_{\mathtt{focal}}$.
- If the project uses `gtest`, retaining only one `main` function in the generated test file; otherwise, ensuring each test method is called by the `main` function.

❷ **Rule-based Compilation Error Fixing**: Following **Phase ❶**, CITYWALK compiles the generated `TC` and applies $\mathtt{Rule}_{\mathtt{F_c}}$ to the failed cases **(Lines 10-11)**. The fixing rules include:
- Fixing incorrect usage of namespaces.
- Deleting non-existent import paths.

❸ **LLM-based Compilation Error Fixing**: If compilation errors persist after **Phase ❷**, CITYWALK transitions to a one-round LLM-based fixing phase **(Lines 12-14)**. During this phase, CITYWALK collects details about the failed test cases along with their associated compilation error messages to construct the fixing prompt. The LLM



You are a highly skilled developer and expert in C++ and testing frameworks. I will provide the content of an entire test file along with the related compilation error messages. Please analyze the situation carefully, identify the causes of the errors, and modify the test file to fix the errors.
// Test File Content:
{test_file}
// Compilation Error Messages:
{error_message}

Fig. 6. The Three-Phase Fixing Process for the LLM-Generated Unit Test Cases.

is then prompted to analyze the root cause of the errors and make corrections to the failed test cases. If any test case continues to fail compilation after this phase, it will be removed from TC.

3 EXPERIMENTAL SETUP

3.1 Research Questions

To assess the effectiveness of CITYWALK, we raise the following two research questions (RQs):

- **RQ1: How does** CITYWALK **perform in C++ unit test generation when compared to state-of-the-art baselines?** This RQ aims to evaluate the superior effectiveness of CITYWALK in comparison to open-source code LLMs, closed-source general-purpose LLMs, and LLM-based unit test generation approaches within the context of C++ unit test generation. To achieve this, we conduct a comprehensive evaluation of CITYWALK against seven baselines using a collection of ten C++ projects. Furthermore, we assess CITYWALK's generalization capabilities using three additional LLM baselines, thereby enhancing evaluation diversity.
- **RQ2: How does each component impact the performance of** CITYWALK**?** Since CITYWALK introduces a step-by-step prompting strategy and leverages various contextual information and post-processing techniques as guidance, this RQ seeks to analyze the contributions of each component through an ablation study.

3.2 Benchmark

To comprehensively evaluate the quality of LLM-generated test cases, we construct a new benchmark consisting of ten real-world open-source C++ projects from GitHub. As shown in Table 3, the first four projects are widely utilized in recent studies on automated C++ unit testing [15, 26, 27]. Additionally, we crawl two practical projects related to basic software: `ninja` (build system) and `leveldb` (database). To mitigate potential data leakage concerns, we include four projects—`json.cpp`, `glomap`, `papy`, and `mlx`—all created after the GPT-4o knowledge cutoff date (October 2023). This ensures that GPT-4o was not trained on these projects. The selection criteria for these projects are as follows. First, each project has received more than 50 stars on GitHub and is actively maintained, ensuring ongoing community interest and updates. Second, the projects span a range of application domains, including parsers (e.g., XML or YAML), basic software systems, and algorithm libraries. Third, the projects feature a number of complex methods with a cyclomatic complexity [20] greater than 10, which suggests the presence of nested control flows, while also covering intricate C++ language features. The size of the selected projects varies from 1.9K lines of code (LoC) to 149.4K LoC. Larger projects are excluded to effectively manage token costs within our limited budget. In total, CITYWALK generates C++ unit test cases for 1288 focal methods across these ten projects. These selection criteria ensure that our benchmark is both high-quality and reproducible, while maintaining a balance between diversity and resource constraints.

Table 3. Statistics of the Collected Open-Source C++ Projects

Project	Application Type	GitHub Stars	Size (LoC)	# Files	# Focal Methods	# Complex Methods	Trained?
hjson-cpp[4]	User Interface for JSON	73	2911	4	25	9 (36.0%)	Y
tinyxml2[5]	XML Parser	5.4K	3606	1	158	5 (3.2%)	Y
yaml-cpp[6]	YAML Parser and Emitter	5.6K	8800	28	204	31 (15.2%)	Y
re2[7]	Regular Expression Engine	9.4K	20373	10	146	41 (28.1%)	Y
ninja[8]	Build System	12.2K	37512	19	238	38 (16.0%)	Y
leveldb[9]	Key-Value Storage Library	38K	149371	17	220	6 (2.7%)	Y
json.cpp[10]	JSON Parsing Library	747	62677	1	34	4 (11.8%)	N
glomap[11]	Map Management Library	1.9K	8477	9	32	4 (12.5%)	N
papy[12]	JSON Data Generator	62	1869	6	25	2 (8.0%)	N
mlx[13]	Array Framework	21.9K	20137	11	206	18 (8.7%)	N

3.3 Baselines

This paper focuses on addressing the C++ unit test generation task using LLMs. To this end, we compare CITYWALK against seven state-of-the-art baselines. First, we select two representative open-source code LLMs: **CodeGeeX4** [56] and **DeepSeek-V3** [10], both of which have demonstrated competitive performance on recent public benchmarks related to coding tasks, such as BigCodeBench [57] and NaturalCodeBench [52]. The selection of these two models is motivated by their distinct characteristics in terms of parameter size (CodeGeeX4 has 9B parameters, while DeepSeek-V3 comprises 671B parameters) and architecture (CodeGeeX4 utilizes a Transformer-based architecture, whereas DeepSeek-V3 employs a Mixture of Experts-based architecture). We exclude CodeLLaMA [28] from our evaluation due to its limited context length. Additionally, we include the closed-source general-purpose LLMs (i.e., **GPT-3.5** [21] and **GPT-4o** [22]) because of their established effectiveness across a broad spectrum of tasks. Finally, we compare CITYWALK with three LLM-based unit test generation approaches for Java and JavaScript: **ChatTester** [51], **HITS** [45], and TestPilot

[4]https://github.com/hjson/hjson-cpp
[5]https://github.com/leethomason/tinyxml2
[6]https://github.com/jbeder/yaml-cpp
[7]https://github.com/google/re2
[8]https://github.com/ninja-build/ninja
[9]https://github.com/google/leveldb
[10]https://github.com/jart/json.cpp
[11]https://github.com/colmap/glomap
[12]https://github.com/noahpop77/Papy
[13]https://github.com/ml-explore/mlx

[33]. Since existing automated C++ unit test generation tools, such as Coyote [27] and CITRUS [15], are currently unavailable, we are unable to reproduce the results reported in their respective papers. Therefore, we do not include them as baselines in this paper.

3.4 Metrics

As illustrated in Algorithm 1, CITYWALK generates a test file for one focal method at a time. It executes the generated files individually and utilizes `llvm-cov` to compute coverage for each focal method. Specifically, this paper employs four evaluation metrics commonly used in existing studies [6, 49, 51] to compare the performance of CITYWALK with the baselines. These metrics collectively provide a comprehensive perspective on the quality, completeness, and effectiveness of the LLM-generated test cases.

- **Compilation Success Rate (CSR)**: This metric represents the percentage of LLM-generated test cases that compile successfully relative to the total test case number.
- **Execution Pass Rate (EPR)**: This metric quantifies the percentage of LLM-generated test cases that pass during execution, reflecting the proportion of tests that yield expected outcomes.
- **Line Coverage ($\mathbf{Cov_L}$)**: This metric assesses the percentage of source code lines within the focal methods that are executed by LLM-generated test cases.
- **Branch Coverage ($\mathbf{Cov_B}$)**: This metric evaluates the percentage of logical conditions in the source code that are explored by LLM-generated test cases.

3.5 Implementation

We implement the core logic of CITYWALK in Python, invoking GPT-4o via its API, specifically using the **gpt-4o** version from the GPT family of models, which is recognized as the most advanced model currently available. We do not truncate input prompts, as the selected LLMs can effectively handle lengthy inputs. We set a maximum output limit of 4096 tokens. In all experiments, we utilize greedy decoding to generate responses, with CITYWALK producing the top-1 chat completion choice for each input prompt. To enhance response stability, we set the sampling temperature to 0. Additionally, we conduct experiments in a zero-shot setting, where no task examples are provided, thereby demonstrating the superiority of CITYWALK.

4 RESULTS AND ANALYSIS

4.1 Answering RQ1

To answer this question, we conduct a comprehensive comparison of CITYWALK against seven baselines using the collected benchmark. For each LLM, we utilize its inference API for implementation, employing the same basic prompt described in Section 1. Specifically, we employ the **gpt-3.5-turbo-0125** version of the GPT-3.5 model. For ChatTester [51], HITS [45], and TestPilot [33], our implementation is based on their open-source reproduction artifacts from GitHub. Furthermore, we apply the same configuration settings as those used in CITYWALK for fair comparison[14].

4.1.1 Experimental Metric Evaluation. Table 4, Table 5, Table 6, and Table 7 present the performance of CITYWALK and selected baselines in C++ unit test generation, evaluated across both correctness metrics (i.e., **CSR** and **EPR**) and

[14]Note that the maximum prompt length used in our experiments (14733 tokens) remains within the context window limits of all evaluated LLMs—GPT-3.5 (16K), CodeGeeX4 (128K), DeepSeek-V3 (128K), and GPT-4o (128K). Therefore, no prompt truncation is necessary during any of the experiments.

Table 4. Comparison of CITYWALK against the Baselines in Terms of **Compilation Success Rate (CSR)**

Project	hjson-cpp	tinyxml2	yaml-cpp	re2	ninja	leveldb	json.cpp	glomap	papy	mlx	**Avg.**
CodeGeeX4	0.00%	6.95%	38.98%	0.09%	4.22%	11.75%	10.70%	35.75%	50.00%	51.35%	20.98%
DeepSeek-V3	0.00%	0.00%	17.79%	0.00%	7.35%	13.55%	25.14%	51.93%	18.75%	68.34%	20.29%
GPT-3.5	0.00%	0.00%	19.34%	5.45%	5.00%	9.64%	31.16%	28.57%	52.22%	72.80%	22.42%
GPT-4o	16.17%	41.26%	22.01%	0.00%	6.90%	9.74%	51.62%	49.70%	47.90%	73.07%	31.84%
ChatTester	20.00%	18.02%	6.42%	4.68%	11.36%	0.00%	18.75%	8.82%	20.00%	18.01%	12.61%
HITS	1.10%	5.94%	5.65%	0.00%	20.37%	2.66%	0.00%	1.92%	0.00%	0.00%	3.76%
TestPilot	26.67%	69.23%	20.47%	0.00%	14.22%	26.18%	70.68%	28.00%	54.17%	2.66%	31.23%
CITYWALK	**100.00%**	**97.25%**	**80.28%**	**64.85%**	**70.09%**	**47.83%**	**100.00%**	**96.70%**	**84.29%**	**92.61%**	**83.39%**

Table 5. Comparison of CITYWALK against the Baselines in Terms of **Execution Pass Rate (EPR)**

Project	hjson-cpp	tinyxml2	yaml-cpp	re2	ninja	leveldb	json.cpp	glomap	papy	mlx	**Avg.**
CodeGeeX4	0.00%	6.00%	2.76%	0.09%	3.82%	0.36%	8.96%	15.54%	40.54%	41.89%	12.00%
DeepSeek-V3	0.00%	0.00%	13.78%	0.00%	6.69%	12.02%	24.02%	39.91%	18.75%	66.29%	18.15%
GPT-3.5	0.00%	0.00%	18.37%	3.41%	4.04%	8.66%	30.82%	10.48%	38.89%	69.50%	18.42%
GPT-4o	14.37%	38.46%	15.75%	0.00%	6.57%	7.61%	49.10%	47.88%	46.22%	69.94%	29.59%
ChatTester	12.00%	13.95%	4.91%	1.75%	8.77%	0.00%	9.38%	8.82%	17.14%	13.27%	9.00%
HITS	1.10%	5.45%	4.52%	0.00%	14.90%	1.73%	0.00%	1.92%	0.00%	0.00%	2.96%
TestPilot	6.67%	64.55%	18.14%	0.00%	13.33%	23.98%	70.30%	12.00%	34.72%	2.66%	24.64%
CITYWALK	**77.50%**	**88.24%**	**67.82%**	**49.37%**	**62.12%**	**41.00%**	**98.91%**	**78.02%**	**81.43%**	**89.13%**	**73.35%**

Table 6. Comparison of CITYWALK against the Baselines in Terms of **Line Coverage ($\mathbf{Cov_L}$)**

Project	hjson-cpp	tinyxml2	yaml-cpp	re2	ninja	leveldb	json.cpp	glomap	papy	mlx	**Avg.**
CodeGeeX4	0.00%	15.30%	16.54%	0.00%	8.44%	10.45%	7.14%	21.75%	7.22%	8.11%	9.50%
DeepSeek-V3	0.00%	0.00%	31.50%	0.00%	20.94%	22.23%	37.31%	30.04%	27.78%	16.80%	18.66%
GPT-3.5	0.00%	0.00%	25.59%	0.84%	10.68%	14.73%	12.09%	16.75%	22.10%	17.60%	12.04%
GPT-4o	7.73%	43.21%	30.17%	0.00%	10.64%	15.45%	26.50%	22.31%	22.89%	17.85%	19.68%
ChatTester	1.68%	45.00%	0.00%	3.05%	17.52%	0.00%	3.56%	4.47%	9.93%	4.58%	8.98%
HITS	0.00%	35.64%	25.56%	0.00%	19.55%	8.53%	0.00%	4.15%	7.02%	0.00%	9.34%
TestPilot	0.00%	70.90%	0.00%	0.00%	0.00%	24.33%	48.17%	0.00%	22.11%	4.55%	17.01%
CITYWALK	**27.94%**	**77.61%**	**44.46%**	**42.56%**	**56.60%**	**25.25%**	**50.71%**	**42.15%**	**53.11%**	**24.20%**	**44.46%**

coverage metrics (i.e., $\mathbf{Cov_L}$ and $\mathbf{Cov_B}$). The best result for each metric is highlighted in **bold**, while the second-best result is underlined. Our experiments yield the following key findings:

(1) CITYWALK **demonstrates superior performance compared to state-of-the-art baselines on the collected benchmark.** When compared to the seven baselines, CITYWALK achieves the best results across all evaluation metrics. A closer examination of the results reveals that the second-best outcomes for each metric vary depending on the project under test. Nevertheless, GPT-4o consistently outperforms other baselines regarding average scores (listed in the **Avg.** rows) of all the four evaluation metrics. Specifically, CITYWALK surpasses the best baseline, GPT-4o, by 51.55% in **CSR**, 43.76% in **ERP**, 24.78% in $\mathbf{Cov_L}$, and 21.55% in $\mathbf{Cov_B}$. These improvements underscore the effectiveness of CITYWALK in the C++ unit test generation task.

(2) CITYWALK **effectively generates fewer compilation errors than the selected baselines.** As observed from Table 4, it is important to that, in many instances, the selected baselines fail to generate any compilable unit test

Table 7. Comparison of CITYWALK against the Baselines in Terms of **Branch Coverage ($\mathbf{Cov_B}$)**

Project	hjson-cpp	tinyxml2	yaml-cpp	re2	ninja	leveldb	json.cpp	glomap	papy	mlx	**Avg.**
CodeGeeX4	0.00%	9.97%	15.73%	0.00%	9.18%	7.88%	10.81%	22.22%	1.35%	2.70%	7.98%
DeepSeek-V3	0.00%	0.00%	24.24%	0.00%	19.03%	23.24%	32.58%	28.89%	15.78%	13.20%	15.70%
GPT-3.5	0.00%	0.00%	21.61%	0.51%	10.22%	12.50%	13.06%	13.33%	8.82%	13.32%	9.34%
GPT-4o	8.82%	37.18%	23.54%	0.00%	8.97%	14.06%	24.84%	17.78%	12.87%	13.40%	16.15%
ChatTester	0.89%	38.45%	0.00%	2.24%	15.46%	0.00%	1.61%	2.48%	4.23%	2.27%	6.76%
HITS	0.00%	26.74%	17.70%	0.00%	16.46%	8.72%	0.00%	3.09%	0.00%	0.00%	7.27%
TestPilot	0.00%	57.14%	0.00%	0.00%	0.00%	21.48%	45.48%	0.00%	8.83%	2.27%	13.52%
CITYWALK	**24.29%**	**65.66%**	**35.56%**	**35.64%**	**49.72%**	**24.64%**	**46.61%**	**37.82%**	**38.97%**	**18.05%**	**37.70%**

Table 8. Comparison of CITYWALK against the Baselines in Terms of the Number of Generated Test Cases

Project	hjson-cpp	tinyxml2	yaml-cpp	re2	ninja	leveldb	json.cpp	glomap	papy	mlx	**Total**	**Failed**
CodeGeeX4	175	417	652	1133	1231	1115	402	193	74	439	5831	5456 (93.57%)
DeepSeek-V3	463	1405	1147	1466	2273	1572	533	233	160	482	9734	8664 (89.01%)
GPT-3.5	59	172	827	587	1139	1235	292	105	90	458	4964	4185 (84.30%)
GPT-4o	167	143	750	777	1218	986	277	165	119	479	5081	4124 (81.16%)
ChatTester	25	172	265	171	308	163	32	34	35	211	1416	1306 (92.23%)
HITS	181	404	1239	1190	1389	752	233	156	78	1033	6655	6352 (95.45%)
TestPilot	15	299	215	94	225	955	266	25	72	188	2354	1642 (69.75%)
CITYWALK	40	255	289	140	565	922	183	91	70	230	2785	1021 (36.66%)

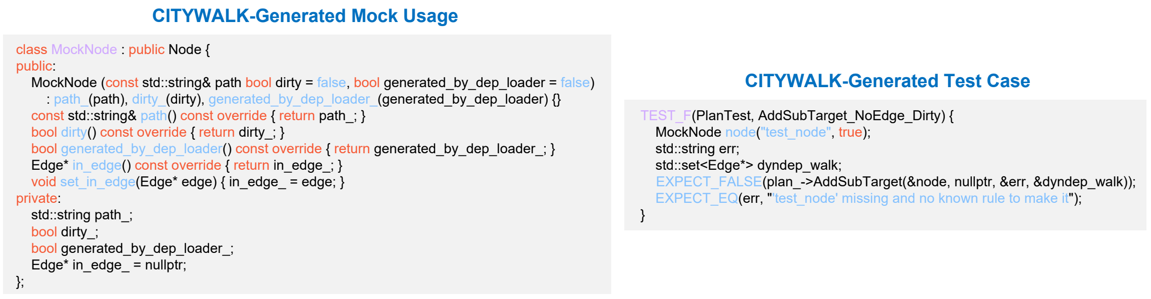
cases for the projects under test using the basic prompt. This issue arises because projects (e.g., `hjson-cpp`) do not include the `gtest` testing framework. Consequently, when the LLMs generate test code without accounting for these configuration dependencies, they default to using the `gtest` framework, resulting in compilation errors for all generated test cases. If configuration-dependent information is incorporated into other baseline approaches, a portion of their compilation errors could be resolved and their performance could potentially be improved. In contrast, the higher **CSR** achieved by CITYWALK is largely due to the inclusion of additional project dependencies. CITYWALK guides the LLMs to generate syntactically correct test code by directly incorporating relevant configuration dependency information into the prompt, thereby greatly reducing the occurrence of compilation errors.

(3) **The project's complexity significantly affects the ability of** CITYWALK **to generate correct test cases with high coverage.** According to the statistical results in the seventh column of Table 3, `tinyxml2` exhibits a smaller proportion of complex methods. Consequently, the test cases generated by CITYWALK for `tinyxml2` achieve both high syntactical correctness, with a **CSR** exceeding 95%, and successful execution, with an **EPR** above 85%. This leads to strong coverage metrics, with both $\mathbf{Cov_L}$ and $\mathbf{Cov_B}$ exceeding 65%. Conversely, for the more complex project `re2`, which contains the most complex methods, the correctness of the generated test code is comparatively low, and the coverage metrics are also less satisfactory. Additionally, although `leveldb` has relatively few complex methods, CITYWALK performs poorly in generating test cases for this project. This underperformance can be attributed to several factors. We will discuss the intricate cases from `leveldb` in Section 4.1.2.

(4) CITYWALK **achieves high code coverage with fewer test cases than the baselines.** Table 8 reports the number of test cases generated by CITYWALK and the baselines for each project. The **Total** column represents

Table 9. Comparison of CITYWALK against the Baselines in Terms of the Complexity and Mock Frequency of Generated Test Cases

Approach	**CodeGeeX4**	**DeepSeek-V3**	**GPT-3.5**	**GPT-4o**	**ChatTester**	**HITS**	TestPilot	CITYWALK
Avg. Size	7.7	7.4	6.8	8.0	7.3	5.9	7.1	9.5
Avg. CC	1.3	1.1	1.1	1.2	1.3	1.0	1.4	1.5
Mock Frequency	26.93%	25.38%	5.60%	25.15%	15.23%	21.44%	4.19%	41.73%



CITYWALK-Generated Mock Usage

```
class MockNode : public Node {
public:
   MockNode (const std::string& path bool dirty = false, bool generated_by_dep_loader = false)
      : path_(path), dirty_(dirty), generated_by_dep_loader_(generated_by_dep_loader) {}
   const std::string& path() const override { return path_; }
   bool dirty() const override { return dirty_; }
   bool generated_by_dep_loader() const override { return generated_by_dep_loader_; }
   Edge* in_edge() const override { return in_edge_; }
   void set_in_edge(Edge* edge) { in_edge_ = edge; }
private:
   std::string path_;
   bool dirty_;
   bool generated_by_dep_loader_;
   Edge* in_edge_ = nullptr;
};
```

CITYWALK-Generated Test Case

```
TEST_F(PlanTest, AddSubTarget_NoEdge_Dirty) {
   MockNode node("test_node", true);
   std::string err;
   std::set<Edge*> dyndep_walk;
   EXPECT_FALSE(plan_->AddSubTarget(&node, nullptr, &err, &dyndep_walk));
   EXPECT_EQ(err, "'test_node' missing and no known rule to make it");
}
```

Fig. 7. Illustrative Test Case with Compliant Mock Usage.

the overall number of test cases generated across all projects. The statistical results indicate that CITYWALK generates fewer test cases compared to the selected baselines (except **ChatTester** and TestPilot), proving that CITYWALK does not rely on sampling a large number of test cases to achieve advantages. The reason **ChatTester** generates the fewest test cases lies in its prompt design, which explicitly instructs the LLM to "*write one test case for each focal method*". This constraint significantly limits the number of test cases generated by **ChatTester**, resulting in suboptimal performance of **ChatTester** across many projects. Furthermore, we examine the number of failed test cases generated by CITYWALK and the baselines, as reported in the **Failed** column of Table 8. The results show that CITYWALK produces significantly fewer failed test cases compared to the baselines, with only 36.66% of its generated test cases failing to compile or execute. This finding suggests that CITYWALK is more effective in generating syntactically and semantically correct test cases while maintaining high code coverage.

(5) **Comparison of the code complexity and mock usage between** CITYWALK **and the baselines.** As shown in Table 9, the rows **Avg. Size** and **Avg. CC** represent the average number of source code lines and the average cyclomatic complexity of the generated test cases, respectively. Overall, the test cases generated by CITYWALK exhibit slightly greater scale and complexity compared to the baselines. This is expected, as generating correct test assertions necessitates a certain level of complex code logic. In addition, we examine the frequency of mock usage, defined as the proportion of test files that employ mock objects. CITYWALK demonstrates a significantly higher mock usage frequency of 41.73%, indicating its greater capability in producing test cases that involve mocking. This is particularly important for effectively testing complex methods and achieving comprehensive code coverage. Beyond the quantitative analysis, we assess the adherence of the generated mocks to established best practices [38]. Figure 7 presents a CITYWALK-generated test case for `Plan::AddSubTarget` from the `ninja` project. In this example, the core logic of the `Plan` module is retained, while its dependency nodes and edges are replaced with mocks to simulate various conditions such as dirty or ready states. This strategy enables precise and isolated testing without compromising behavioral integrity or introducing excessive mocking.

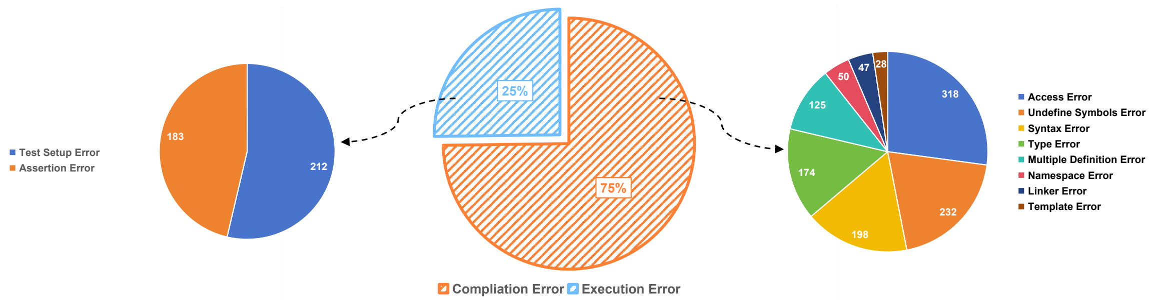


Fig. 8. Error Category and Frequency.

(6) **The correctness and coverage metrics do not necessarily indicate a positive correlation.** As illustrated in Table 4, Table 5, Table 6, and Table 7, high correctness does not always guarantee high code coverage, and low correctness can sometimes lead to higher coverage. For instance, **DeepSeek-V3** on `yaml-cpp` demonstrates this phenomenon. Despite achieving lower **CSR** and **EPR** compared to other LLM baselines, **DeepSeek-V3** attains the second-highest code coverage scores. According to the fourth column in Table 8, it is evident that **DeepSeek-V3** generates more test cases for `yaml-cpp` than other LLM baselines. Consequently, the proportion of incorrect test cases is also relatively high, leading to a scenario characterized by low correctness but high coverage.

(7) **The two LLM-based unit test generation approaches for Java (i.e., ChatTester and HITS) achieve low performance on the collected C++ projects.** First, the output of LLMs can vary significantly depending on the prompt design. If the prompt does not account for the specific characteristics of different programming languages, the performance of existing Java-specific baselines on C++ unit test generation may not generalize effectively. Furthermore, to ensure fairness in our experiments, all baselines that use LLMs for post-processing perform only a single round of iterative fixes, similar to CITYWALK, which is much fewer than the number of iterations used in their original papers. As a result, the reproduction performance may be lower than the corresponding results reported. However, this also suggests that CITYWALK is less dependent on iterative fixes to achieve its performance.

4.1.2 Bad Case Breakdown. We further conduct an in-depth investigation into the common error categories of the failed test cases generated by CITYWALK. Specifically, we manually analyze the output error messages associated with these failed test cases and summarize the corresponding error categories. In total, CITYWALK produces 1567 errors across 1021 failed test cases in all ten projects. Since a single test case can contain multiple errors, the total error count exceeds the number of failed test cases. Our analysis reveals the following findings based on the statistical results:

(1) **The two most prevalent categories of errors in the failed test cases are compilation errors and execution errors.** As illustrated in Figure 8, we subdivide the two high-level categories into distinct sub-categories. The most common sub-category of compilation errors is **Access Error**, which occurs when LLMs frequently generate test code that attempts to invalidly access private variables or methods. The most common sub-category of execution errors is **Test Setup Error**, primarily due to LLMs facing challenges in accurately inferring the mock object configurations, as well as generating valid test data inputs.

Table 10. Generalization Results of CITYWALK on Different LLMs in Terms of **Compilation Success Rate (CSR)**

Project	hjson-cpp	tinyxml2	yaml-cpp	re2	ninja	leveldb	json.cpp	glomap	papy	mlx	Avg. ↑
CodeGeeX4	0.00%	6.95%	38.98%	0.09%	4.22%	11.75%	10.70%	35.75%	50.00%	51.35%	**19.76%**
CodeGeeX4 w/ CITYWALK	43.59%	42.11%	54.64%	4.30%	31.45%	29.12%	36.45%	42.44%	61.11%	62.22%	
DeepSeek-V3	0.00%	0.00%	17.79%	0.00%	7.35%	13.55%	25.14%	51.93%	18.75%	68.34%	**37.94%**
DeepSeek-V3 w/ CITYWALK	74.62%	72.67%	43.45%	47.83%	41.56%	42.53%	53.89%	62.34%	70.11%	73.24%	
GPT-3.5	0.00%	0.00%	19.34%	5.45%	5.00%	9.64%	31.16%	28.57%	52.22%	72.80%	**23.66%**
GPT-3.5 w/ CITYWALK	31.78%	72.96%	33.69%	20.05%	27.53%	32.45%	45.23%	37.34%	86.67%	73.07%	

Table 11. Generalization Results of CITYWALK on Different LLMs in Terms of **Execution Pass Rate (EPR)**

Project	hjson-cpp	tinyxml2	yaml-cpp	re2	ninja	leveldb	json.cpp	glomap	papy	mlx	Avg. ↑
CodeGeeX4	0.00%	6.00%	2.76%	0.09%	3.82%	0.36%	8.96%	15.54%	40.54%	41.89%	**21.36%**
CodeGeeX4 w/ CITYWALK	30.77%	37.92%	42.92%	2.15%	26.78%	25.56%	21.78%	32.34%	55.56%	57.78%	
DeepSeek-V3	0.00%	0.00%	13.78%	0.00%	6.69%	12.02%	24.02%	39.91%	18.75%	66.29%	**33.43%**
DeepSeek-V3 w/ CITYWALK	67.69%	64.92%	31.82%	42.61%	36.78%	37.63%	47.12%	51.56%	65.52%	70.12%	
GPT-3.5	0.00%	0.00%	18.73%	3.41%	4.04%	8.66%	30.82%	10.48%	38.89%	69.50%	**23.71%**
GPT-3.5 w/ CITYWALK	31.78%	70.24%	28.75%	12.98%	32.89%	28.72%	38.47%	27.45%	80.00%	69.94%	

(2) **The complexity of C++ language features poses significant challenges for** CITYWALK **in generating accurate test cases.** In the design of CITYWALK, we integrate language-specific domain knowledge derived from empirical observations to address common errors, thereby enhancing the performance of the LLM to some extent. However, since the benchmark is collected from a diverse range of real-world projects, project-specific issues may still impede the generation of correct and high-coverage test code. For instance, many functionalities in the large basic software system `leveldb` rely heavily on external interactions, such as disk I/O operations and file content checks. This dependence often compels LLMs to redefine existing methods in dependent classes or to invoke undefined methods when constructing I/O streams, leading to errors such as **Multiple Definition Error** and **Undefined Symbol Error**, which result in low evaluation metrics. Additionally, `leveldb` involves advanced features like synchronization locks, which significantly increase the complexity and challenge the test generation process. Moreover, the absence of an automated and strict exception handling mechanism, akin to that in Java, complicates the LLM's ability to generate correct test code with exception handling logic. The most frequently occurring assertion errors can be traced back to the lack of supplementary post-processing techniques (e.g., post-hoc fix), which could help prompt LLMs to rectify incorrect assertions in the generated test cases. We are concerned that providing LLMs with error messages that include expected outputs may encourage them to replicate those outputs directly, thus manipulating the compiler into successful execution of the generated test cases. We will discuss these *false positive* cases in detail in Section 5.4.

4.1.3 Generalizability Evaluation. We further employ CITYWALK to three LLMs, including CodeGeeX4, DeepSeek-V3, and GPT-3.5. Specifically, we conduct ablation experiments on each LLM individually to investigate the impact of querying the corresponding LLM using the prompting strategy and additional contextual guidance designed by CITYWALK. Table 10, Table 11, Table 12, and Table 13 present the comparison results using the correctness and coverage metrics, respectively. Each LLM's results are displayed in two lines: the first line shows the results when the LLM directly utilizes the basic prompt to generate unit test cases for the given focal methods, and the second line presents the results when integrated with CITYWALK. We derive two key insights from the statistical results: ① **Integrating** CITYWALK

Table 12. Generalization Results of CITYWALK on Different LLMs in Terms of **Line Coverage ($\mathbf{Cov_L}$)**

Project	hjson-cpp	tinyxml2	yaml-cpp	re2	ninja	leveldb	json.cpp	glomap	papy	mlx	Avg. ↑
CodeGeeX4	0.00%	15.30%	16.54%	0.00%	8.44%	10.45%	7.14%	21.75%	7.22%	8.11%	**14.42%**
CodeGeeX4 w/ CITYWALK	15.78%	46.15%	35.28%	28.04%	19.12%	12.45%	26.89%	37.89%	8.11%	9.44%	
DeepSeek-V3	0.00%	0.00%	31.50%	0.00%	20.94%	22.23%	37.31%	30.04%	27.78%	16.80%	**21.79%**
DeepSeek-V3 w/ CITYWALK	22.44%	78.17%	42.80%	39.71%	52.23%	26.63%	48.35%	43.45%	32.87%	17.85%	
GPT-3.5	0.00%	0.00%	25.59%	0.84%	10.68%	14.73%	12.09%	16.75%	22.10%	17.60%	**21.67%**
GPT-3.5 w/ CITYWALK	10.01%	74.87%	42.59%	30.62%	32.45%	20.12%	23.56%	32.56%	52.43%	17.85%	

Table 13. Generalization Results of CITYWALK on Different LLMs in Terms of **Branch Coverage ($\mathbf{Cov_B}$)**

Project	hjson-cpp	tinyxml2	yaml-cpp	re2	ninja	leveldb	json.cpp	glomap	papy	mlx	Avg. ↑
CodeGeeX4	0.00%	9.97%	15.73%	0.00%	9.18%	7.88%	10.81%	22.22%	1.35%	2.70%	**10.93%**
CodeGeeX4 w/ CITYWALK	5.18%	33.07%	31.23%	21.09%	21.45%	10.72%	21.53%	32.98%	5.25%	6.67%	
DeepSeek-V3	0.00%	0.00%	24.24%	0.00%	19.03%	23.24%	32.58%	28.89%	15.78%	13.20%	**19.30%**
DeepSeek-V3 w/ CITYWALK	23.65%	65.98%	35.06%	31.94%	51.89%	25.15%	45.41%	38.12%	18.88%	13.87%	
GPT-3.5	0.00%	0.00%	21.61%	0.51%	10.22%	12.50%	13.06%	13.33%	8.82%	13.32%	**16.86%**
GPT-3.5 w/ CITYWALK	12.70%	61.39%	34.99%	24.66%	27.12%	17.33%	19.84%	27.46%	23.09%	13.40%	

as a complementary plug-in enhances the performance of C++ unit test generation. As observed, the three evaluated LLMs demonstrate consistent improvements in all four metrics across the ten projects. ② **Performance gains scale markedly with parameter count.** For example, the average performance gain of each metric (listed in the **Avg.** ↑ column) with DeepSeek-V3-671B is substantially greater than that of CodeGeeX4-9B.

Answer to RQ1: In conclusion, CITYWALK exhibits a marked superiority over the LLM-based baselines across all evaluation metrics, highlighting its effectiveness in C++ unit test generation. Furthermore, CITYWALK effectively harnesses the latent intelligence of the LLMs, demonstrating the potential for seamless integration with additional LLMs in a plug-and-play manner.

4.2 Answering RQ2

To answer this question, we conduct a series of ablation experiments to assess the impact of different designed components within CITYWALK. To ensure the fairness of comparisons, the parameter configurations align with those described in Section 3.5.

4.2.1 Ablation Study. The components within the CITYWALK design include configuration dependencies $\texttt{Dep}_{\texttt{c}}$, cross-file data dependencies $\texttt{Dep}_{\texttt{d}}$, intention contexts $\texttt{Context}_{\texttt{intent}}$, guidelines of language-specific domain knowledge $\texttt{Guideline}_{\texttt{DK}}$, step-by-step instructions $\texttt{PROMPT}_{\texttt{step}}$, and post-processing techniques $\texttt{Fix}_{\texttt{rule+prompt}}$. Specifically, we perform ablation experiments by removing one component at a time and analyze the performance contribution of each component regarding the coverage metrics $\mathbf{Cov_L}$ and $\mathbf{Cov_B}$. Table 14 and Table 15 show the performance contribution results (denoted as the performance degradation values). The greater the coverage scores decline, the larger the contribution of that component. The **Avg.** ↓ columns present the average results across all projects. When compared to CITYWALK, each ablation model exhibits varying degrees of decline in terms of $\mathbf{Cov_L}$ and $\mathbf{Cov_B}$, indicating that each designed component contributes to the overall enhancement in generating high-quality unit test cases. According to the

Table 14. Performance Contribution of Each Component in Terms of **Line Coverage ($\mathbf{Cov_L}$)**

Project	hjson-cpp	tinyxml2	yaml-cpp	re2	ninja	leveldb	json.cpp	glomap	papy	mlx	**Avg.** ↓
CITYWALK	27.94%	77.61%	44.46%	42.56%	56.60%	25.25%	50.71%	42.15%	53.11%	24.20%	
w/o $\mathtt{Dep_c}$	-27.94%	-77.31%	-12.08%	-42.56%	-3.82%	-1.36%	-50.71%	-1.81%	-53.11%	-24.20%	**-29.49%**
w/o $\mathtt{Dep_d}$	-0.82%	-32.42%	-14.50%	-7.07%	-15.26%	-8.69%	-19.04%	-13.03%	-15.51%	-3.48%	**-12.98%**
w/o $\mathtt{Context_{intent}}$	-12.36%	-29.84%	-8.72%	-8.21%	-11.21%	-5.11%	-6.59%	-9.70%	-11.38%	-0.90%	**-10.40%**
w/o $\mathtt{PROMPT_{step}}$	-5.46%	-12.97%	-12.33%	-9.86%	-8.42%	-8.91%	-12.48%	-7.59%	-16.34%	-7.50%	**-10.19%**
w/o $\mathtt{Guideline_{DK}}$	-16.74%	-7.40%	-11.48%	-7.87%	-7.04%	-2.69%	-3.59%	-5.59%	-24.61%	-6.29%	**-9.33%**
w/o $\mathtt{Fix_{rule+prompt}}$	-5.24%	-12.51%	-14.95%	-8.48%	-10.76%	-1.51%	-1.27%	-4.26%	-7.42%	-3.94%	**-7.03%**

Table 15. Performance Contribution of Each Component in Terms of **Branch Coverage ($\mathbf{Cov_B}$)**

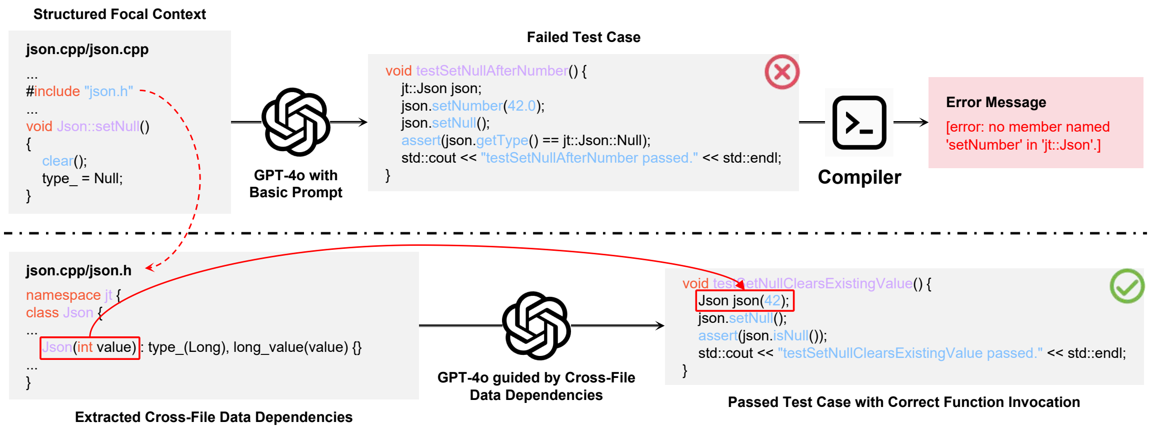
Project	hjson-cpp	tinyxml2	yaml-cpp	re2	ninja	leveldb	json.cpp	glomap	papy	mlx	**Avg.** ↓
CITYWALK	24.29%	65.66%	35.56%	35.64%	49.72%	24.64%	46.61%	37.82%	38.97%	18.05%	
w/o $\mathtt{Dep_c}$	-24.29%	-65.66%	-10.04%	-35.64%	-3.38%	-1.50%	-46.61%	-1.70%	-38.97%	-18.05%	**-24.58%**
w/o $\mathtt{Dep_d}$	-1.99%	-30.69%	-11.48%	-7.94%	-14.60%	-8.75%	-19.33%	-13.93%	-18.61%	-4.92%	**-13.22%**
w/o $\mathtt{Context_{intent}}$	-14.56%	-28.48%	-7.00%	-9.87%	-9.60%	-5.17%	-6.83%	-9.59%	-14.14%	-2.93%	**-10.82%**
w/o $\mathtt{PROMPT_{step}}$	-5.97%	-13.60%	-10.19%	-9.90%	-6.16%	-8.97%	-12.49%	-7.59%	-19.04%	-9.25%	**-10.32%**
w/o $\mathtt{Guideline_{DK}}$	-15.51%	-8.07%	-8.70%	-9.02%	-5.83%	-2.75%	-3.72%	-5.70%	-28.97%	-4.65%	**-9.29%**
w/o $\mathtt{Fix_{rule+prompt}}$	-6.12%	-9.68%	-8.52%	-9.60%	-30.33%	-3.11%	-0.64%	-3.26%	-8.92%	-5.83%	**-8.60%**

statistical results presented in Table 14 and Table 15, the top three components that contribute the most to CITYWALK are $\mathtt{Dep_c}$, $\mathtt{Dep_d}$, and $\mathtt{Context_{intent}}$.

As discussed in Section 4.1.1, $\mathtt{Dep_c}$ plays a crucial role in guiding LLMs to generate syntactically correct test code by directly incorporating configuration dependencies. Thus, projects not utilizing `gtest` (e.g., `tinyxml2` and `papy`) would benefit from $\mathtt{Dep_c}$ as it helps LLMs recognize the absence of this testing framework, thereby preventing hallucinate invocations of frameworks or libraries absent from the project's dependencies. It is worth noting that post-processing techniques (e.g., Pynguin's method injection [18]) could also address certain types of compilation errors. In the design of CITYWALK, we integrate both proactive prevention strategies (e.g., explicitly incorporating $\mathtt{Dep_c}$) and reactive post-hoc fixes (i.e., $\mathtt{Fix_{rule+prompt}}$). Our ablation study reveals that using only post-processing (**w/o** $\mathtt{Dep_c}$) leads to a 29.49% drop in average $\mathbf{Cov_L}$ and a 24.58% drop in average $\mathbf{Cov_B}$. In contrast, using only proactive prevention (**w/o** $\mathtt{Fix_{rule+prompt}}$) results in a smaller decrease: 7.03% in average $\mathbf{Cov_L}$ and 8.60% in average $\mathbf{Cov_B}$. These results indicate that relying solely on post-hoc fixes is less effective than employing proactive prevention alone.

Figure 9 illustrates how extracted cross-file data dependencies are used as guidance for LLMs to generate correct test cases. By providing the key $\mathtt{Dep_d}$ (i.e., the initialization constructor of `Json`) from the `json.h` file, LLMs can prevent compilation errors that would otherwise arise due to invoking the non-existent function `setNumber`. Figure 10 illustrates how $\mathtt{Context_{intent}}$ can guide LLMs in generating correct test cases for the **Failed Test Case ❷** within Figure 1. Specifically, the retrieved `ParseTag` method within the `singledocparser` class provides an invocation example of the focal method, while the retrieved `_Tag` method within the `emittermanip` header file offers an initialization example of the focal method. These examples significantly aid LLMs in understanding the usage patterns of the focal method, contributing to the generation of functionally correct assertions and enhancing code coverage.

4.2.2 The Impact of Different Error-Fixing Phases within CITYWALK. We further investigate the influence of individual error-fixing phases within our three-phase post-processing techniques on test case quality enhancement. As described in Section 2.6.2, Phase ❶ and Phase ❷ focus on resolving syntactic and compilation errors via predefined rules, while


Structured Focal Context
GPT-4o with Basic Prompt
Failed Test Case
Compiler
Error Message
GPT-4o guided by Cross-File Data Dependencies
Extracted Cross-File Data Dependencies
Passed Test Case with Correct Function Invocation

json.cpp/json.cpp

```
...
#include "json.h"
...
void Json::setNull()
{
    clear();
    type_ = Null;
}
```

```
void testSetNullAfterNumber() {
    jt::Json json;
    json.setNumber(42.0);
    json.setNull();
    assert(json.getType() == jt::Json::Null);
    std::cout << "testSetNullAfterNumber passed." << std::endl;
}
```

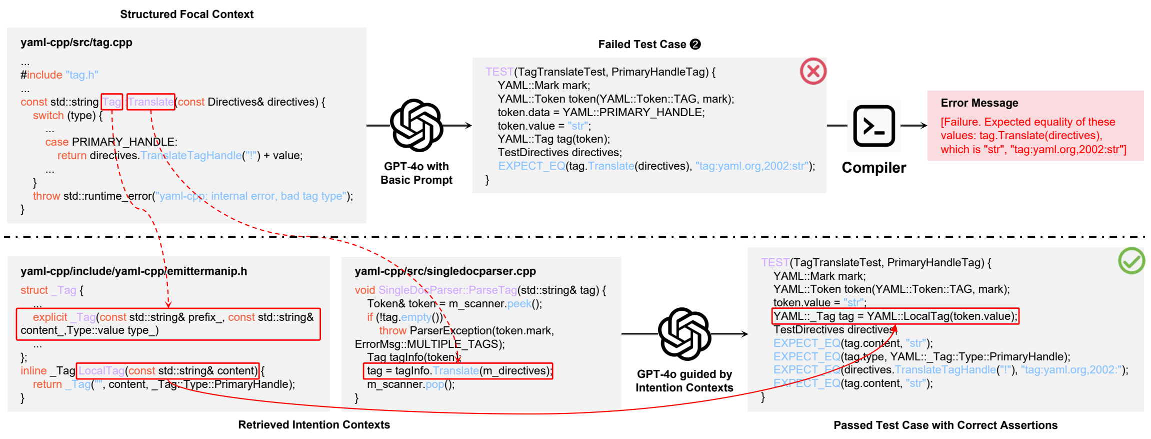
```
[error: no member named 'setNumber' in 'jt::Json'.]
```

json.cpp/json.h

```
namespace jt {
class Json {
...
    Json(int value) : type_(Long), long_value(value) {}
...
}
```

```
void testSetNullClearsExistingValue() {
    Json json(42);
    json.setNull();
    assert(json.isNull());
    std::cout << "testSetNullClearsExistingValue passed." << std::endl;
}
```

Fig. 9. Illustration Example of How Cross-File Data Dependencies Guide LLMs Generate Correct Test Cases.


Structured Focal Context
GPT-4o with Basic Prompt
Failed Test Case ❷
Compiler
Error Message
GPT-4o guided by Intention Contexts
Retrieved Intention Contexts
Passed Test Case with Correct Assertions

yaml-cpp/src/tag.cpp

```
...
#include "tag.h"
...
const std::string Tag::Translate(const Directives& directives) {
    switch (type) {
        ...
        case PRIMARY_HANDLE:
            return directives.TranslateTagHandle("!") + value;
        ...
    }
    throw std::runtime_error("yaml-cpp: internal error, bad tag type");
}
```

```
TEST(TagTranslateTest, PrimaryHandleTag) {
    YAML::Mark mark;
    YAML::Token token(YAML::Token::TAG, mark);
    token.data = YAML::PRIMARY_HANDLE;
    token.value = "str";
    YAML::Tag tag(token);
    TestDirectives directives;
    EXPECT_EQ(tag.Translate(directives), "tag:yaml.org,2002:str");
}
```

```
[Failure. Expected equality of these values: tag.Translate(directives), which is "str", "tag:yaml.org,2002:str"]
```

yaml-cpp/include/yaml-cpp/emittermanip.h

```
struct _Tag {
    ...
    explicit _Tag(const std::string& prefix_, const std::string&
content_,Type::value type_)
    ...
};
inline _Tag LocalTag(const std::string& content) {
    return _Tag("", content, _Tag::Type::PrimaryHandle);
}
```

yaml-cpp/src/singledocparser.cpp

```
void SingleDocParser::ParseTag(std::string& tag) {
    Token& token = m_scanner.peek();
    if (!tag.empty())
        throw ParserException(token.mark,
ErrorMsg::MULTIPLE_TAGS);
    Tag tagInfo(token);
    tag = tagInfo.Translate(m_directives);
    m_scanner.pop();
}
```

```
TEST(TagTranslateTest, PrimaryHandleTag) {
    YAML::Mark mark;
    YAML::Token token(YAML::Token::TAG, mark);
    token.value = "str";
    YAML::_Tag tag = YAML::LocalTag(token.value);
    TestDirectives directives;
    EXPECT_EQ(tag.content, "str");
    EXPECT_EQ(tag.type, YAML::_Tag::Type::PrimaryHandle);
    EXPECT_EQ(directives.TranslateTagHandle("!"), "tag:yaml.org,2002:");
    EXPECT_EQ(tag.content, "str");
}
```

Fig. 10. Illustration Example of How Intention Contexts Guide LLMs Generate Correct Test Cases.

Phase ❸ leverages the LLM to address compilation errors. Through ablation experiments that incrementally incorporate each phase, we quantify their respective contributions using the correctness metrics **CSR** and **EPR**. Table 16 and Table 17 present phase-by-phase performance improvements, where higher correctness score increments reflect greater phase contributions. The **Avg.** ↑ columns aggregate cross-project average results. Specifically, the integration of Phase ❶ yields improvements of 3.10% in CSR and 3.19% in EPR. Subsequent incorporation of Phase ❷ delivers substantial enhancements, achieving additional gains of 29.80% in **CSR** and 25.98% in **EPR**. Including Phase ❸ sustains quality improvements with further increases of 8.65% and 6.75% in **CSR** and **EPR**, respectively. The varying contributions across phases originate from differences in the types of errors resolved. Phase ❶ focuses on correcting basic syntactic errors. However, since CITYWALK already incorporates prompt optimization techniques that significantly reduce such errors, this phase yields only marginal improvements. In contrast, Phase ❷ utilizes compiler feedback to resolve a broader range of rule-based and compiler-diagnosable errors, resulting in more substantial gains. Finally, Phase ❸ targets complex compilation errors that require deeper semantic reasoning about the root causes, thereby further

Table 16. Performance Contribution of Each Error-Fixing Phase in Terms of **Compilation Success Rate (CSR)**

Project	hjson-cpp	tinyxml2	yaml-cpp	re2	ninja	leveldb	json.cpp	glomap	papy	mlx	**Avg.** ↑
No Error-Fixing Phase	77.08%	67.96%	27.59%	34.76%	22.15%	28.78%	74.56%	36.91%	54.37%	22.97%	
w/ Phase ❶	+0.00%	+5.97%	+0.00%	+1.72%	+1.60%	+4.97%	+0.00%	+16.78%	+0.00%	+0.00%	**+3.10%**
w/ Phase ❶ + ❷	+22.92%	+27.97%	+37.32%	+56.04%	+44.52%	+11.39%	+16.58%	+58.33%	+0.00%	+53.90%	**+32.90%**
w/ Phase ❶ + ❷ + ❸	+22.92%	+29.29%	+52.69%	+58.81%	+47.94%	+19.05%	+25.44%	+59.79%	+29.92%	+69.64%	**+41.55%**

Table 17. Performance Contribution of Each Error-Fixing Phase in Terms of **Execution Pass Rate (EPR)**

Project	hjson-cpp	tinyxml2	yaml-cpp	re2	ninja	leveldb	json.cpp	glomap	papy	mlx	**Avg.** ↑
No Error-Fixing Phase	54.17%	61.27%	23.24%	31.33%	19.85%	27.12%	71.93%	27.52%	45.69%	21.70%	
w/ Phase ❶	+4.16%	+7.66%	+0.00%	+1.58%	+0.30%	+3.42%	+0.00%	+14.76%	+0.00%	+0.00%	**+3.19%**
w/ Phase ❶ + ❷	+21.51%	+25.61%	+32.46%	+51.43%	+37.27%	+9.44%	+15.05%	+47.48%	+0.00%	+51.43%	**+29.17%**
w/ Phase ❶ + ❷ + ❸	+23.33%	+26.97%	+44.58%	+55.81%	+42.27%	+13.88%	+26.98%	+50.50%	+7.42%	+67.43%	**+35.92%**

enhancing test case quality. Statistical analysis demonstrates that all three phases exhibit non-negative performance impacts, collectively enhancing test case quality across both metrics.

Answer to RQ2: To sum up, all components of CITYWALK significantly improve the performance of C++ unit test generation in terms of the coverage metrics.

5 DISCUSSION

5.1 Effectiveness of CITYWALK in Bug Detection

Detecting real software bugs is a critical criterion for evaluating the effectiveness of automated unit test generation approaches. To complement our assessment based on correctness and coverage metrics, we employ mutation testing to measure the bug-detection capability of CITYWALK-generated test cases. Prior studies [12, 16] have validated the utility of mutation testing for both evaluating test quality and guiding the generation of more robust test cases. Mutation testing works by introducing small artificial bugs (i.e., **mutants**) into the program under test. A test suite is considered effective if it can distinguish the mutated version from the original, i.e., "kill" the mutant by triggering observable failures. In this study, we adopt the updated open-source tool *universalmutator* [9], which supports a broad range of mutation operations, as outlined below:

- **Arithmetic operator mutations**: e.g., $+ \leftrightarrow -$, $* \leftrightarrow /$
- **Comparison operator mutations**: e.g., $< \leftrightarrow >$, $== \leftrightarrow\ !=$
- **Logical operator mutations**: e.g., && ↔ ||
- **Control structure mutations**: e.g., removing `else`, `break` ↔ `continue`
- **Structural mutations**: e.g., deleting or commenting out code blocks
- **Literal and constant mutations**: e.g., `true` ↔ `false`, `0` ↔ `1`, string substitutions

To facilitate analysis, we select two single-file projects, `tinyxml2` and `json.cpp`, as evaluation subjects. We first perform mutation testing on the test cases generated by CITYWALK, and then calculate the mutation score, which is defined as the ratio of killed mutants to total valid mutants. A higher mutation score indicates stronger bug detection capabilities and higher oracle quality [47]. We yield the following key observations according to the evaluation results presented in Table 18:

Table 18. Mutation Testing for CITYWALK-Generated Test Cases on `tinyxml2` and `json.cpp`

Project	# Total Valid Mutants	# Killed Mutants	Mutation Score
`tinyxml2`	6272	5088	81.12%
`json.cpp`	844	754	89.34%

(1) CITYWALK-generated test cases respective achieve a mutation score of 89.34% for `json.cpp` and 81.12% for `tinyxml2`. These results indicate that CITYWALK is highly effective in generating test cases that detect a substantial proportion of artificial bugs introduced into the code, validating its capability to uncover potential bugs.

(2) The mutation score for `json.cpp` (created after the GPT-4o knowledge cutoff) outperforms `tinyxml2` (which has a risk of potential data leakage) by 8.22%. This provides additional evidence that CITYWALK's ability to generate high-quality test cases is primarily driven by the designed components as guidance, rather than relying on GPT-4o's memorization of training data.

(3) According to the results from the SBST 2022 tool competition [34], EvoSuite achieved a mutation score of 34.1% on the competition benchmark. Additionally, the open-source tool *universalmutator* achieved an average mutation score of 35% on the evaluated C++ projects in the corresponding paper [9]. In comparison, the mutation scores achieved by CITYWALK are notably higher, further emphasizing the effectiveness of CITYWALK-generated test cases.

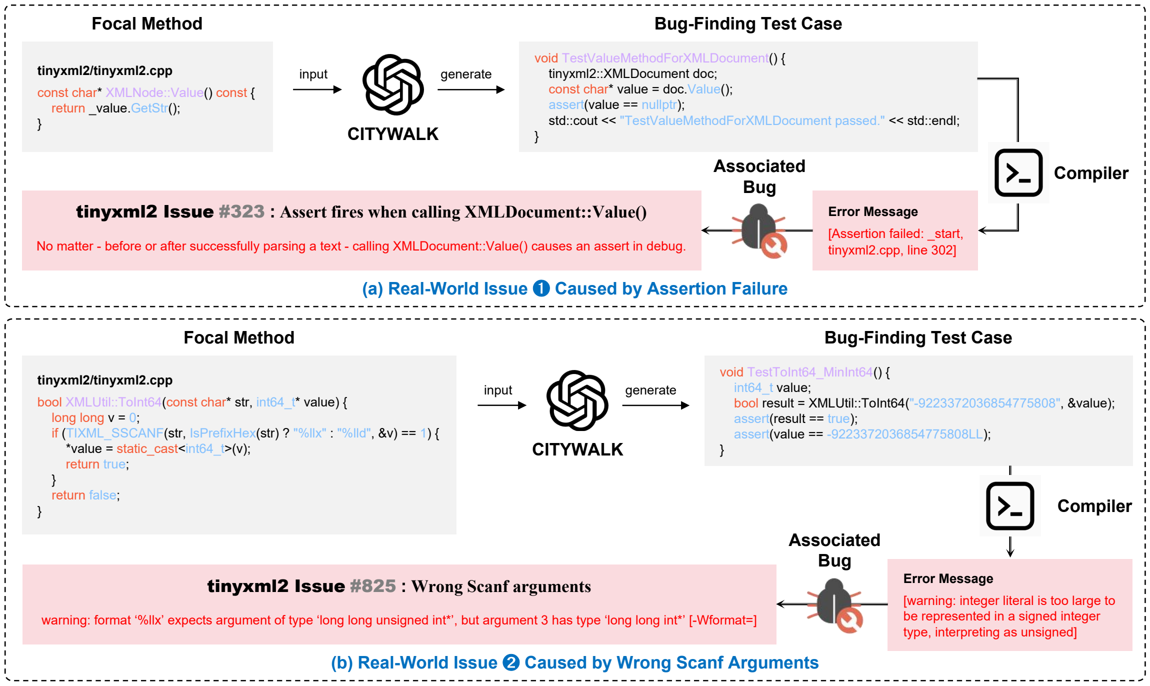
Furthermore, we conduct a case-by-case analysis to assess the usefulness of CITYWALK in discovering real-world bugs. We manually inspect two bugs from `tinyxml2` that were identified by CITYWALK-generated test cases. These are real-world issues that have been reported by human developers. As illustrated in Figure 11(a), the CITYWALK-generated test case first creates an `XMLDocument` object, `doc`, which then invokes the `Value` function. This function, in turn, calls `XMLNode::Value()`. Since the variable `_value` within `XMLNode::Value()` may not have been initialized, it triggers an assertion error in the `GetStr` function. This bug corresponds to **Issue #323**[15] in `tinyxml2`, where a human developer also commits an assertion failure when calling `Value`. Figure 11(b) shows another bug triggered by CITYWALK, where invalid hexadecimal format string arguments are passed to `ToInt64`. This bug is associated with **Issue #825**[16] in `tinyxml2`, where a human developer also identifies a similar issue when calling the static member function `ToInt64`. In summary, the test cases generated by CITYWALK demonstrate the potential to detect real bugs in open-source projects.

5.2 Efficiency of CITYWALK

To evaluate the efficiency and practical feasibility of CITYWALK in the context of C++ unit test generation, Table 19 presents the average execution time per focal method and the corresponding token usage for CITYWALK and each baseline approach. The following insights can be drawn from the statistical results:

(1) Despite not being the fastest or most cost-efficient approach, CITYWALK maintains acceptable computational overhead. The average execution time per focal method is 34.59 seconds, significantly lower than the slowest baseline (**HITS** at 184.39 seconds). Since writing unit tests is time-consuming [13], CITYWALK remains a viable option for developers to enhance testing accuracy while ensuring prompt response times. Moreover, although CITYWALK incurs a higher average token usage (5726 tokens per method) compared to most baselines, the

[15]https://github.com/leethomason/tinyxml2/issues/323
[16]https://github.com/leethomason/tinyxml2/issues/825



tinyxml2/tinyxml2.cpp

```
const char* XMLNode::Value() const {
    return _value.GetStr();
}
```

```
void TestValueMethodForXMLDocument() {
    tinyxml2::XMLDocument doc;
    const char* value = doc.Value();
    assert(value == nullptr);
    std::cout << "TestValueMethodForXMLDocument passed." << std::endl;
}
```

Error Message

[Assertion failed: _start, tinyxml2.cpp, line 302]

tinyxml2 Issue #323 : Assert fires when calling XMLDocument::Value()

No matter - before or after successfully parsing a text - calling XMLDocument::Value() causes an assert in debug.

tinyxml2/tinyxml2.cpp

```
bool XMLUtil::ToInt64(const char* str, int64_t* value) {
    long long v = 0;
    if (TIXML_SSCANF(str, IsPrefixHex(str) ? "%llx" : "%lld", &v) == 1) {
        *value = static_cast<int64_t>(v);
        return true;
    }
    return false;
}
```

```
void TestToInt64_MinInt64() {
    int64_t value;
    bool result = XMLUtil::ToInt64("-9223372036854775808", &value);
    assert(result == true);
    assert(value == -9223372036854775808LL);
}
```

Error Message

[warning: integer literal is too large to be represented in a signed integer type, interpreting as unsigned]

tinyxml2 Issue #825 : Wrong Scanf arguments

warning: format '%llx' expects argument of type 'long long unsigned int*', but argument 3 has type 'long long int*' [-Wformat=]

Fig. 11. Illustration of Two Real-World Issues from `tinyxml2` Found by CITYWALK.

incremental cost per method is approximately $0.03 higher than the best baseline TestPilot, which may be justifiable given CITYWALK's improved correctness and coverage—especially in scenarios where test reliability is prioritized (e.g., safety-critical systems).

(2) Although CITYWALK does not achieve the lowest values in terms of efficiency metrics, it consistently delivers superior test quality, stemming from the designed contextual guidance and post-processing mechanisms. This indicates that CITYWALK strategically trades a moderate increase in computational and monetary cost for substantial gains in test effectiveness, a trade-off often justified in practical software development scenarios.

Table 19. Efficiency Comparison of CITYWALK against the Baselines

Approach	CodeGeeX4	DeepSeek-V3	GPT-3.5	GPT-4o	ChatTester	HITS	TestPilot	CITYWALK
Avg. Execution Time (s)	15.93	57.02	10.26	18.62	70.30	184.39	24.45	34.59
Avg. Token Usage	2533	3553	1849	2583	2222	11736	1436	5726

5.3 Readability and Usability of Test Cases Generated by CITYWALK

The ultimate goal of automated unit test generation is to assist developers in writing test cases. To compare the readability and usability of the test cases generated by CITYWALK with those produced by four LLM-based baselines (i.e., **CodeGeeX4**, **DeepSeek-V3**, **GPT-3.5**, and **GPT-4o**), we conduct a human evaluation to determine developer preference. We invite five participants, each with over three years of C++ development experience, to perform the assessment. We focus on focal methods for which both CITYWALK and the LLMs generated correct test cases, as recommending test cases that fail to compile or execute is impractical. We further limit our analysis to methods with existing human-written test cases in the original project repositories. Consequently, we randomly select 25 focal methods across five projects from our benchmark. Each participant is required to evaluate 150 test cases, comprising one CITYWALK-generated, four LLM-generated, and one human-written test case for each focal method. Each test case is independently scored across the following four aspects. Scores of each aspect range from 1 (lowest quality) to 3 (highest quality). To avoid bias, participants are not informed about the source of each test case (i.e., whether it is LLM-generated or human-written).

- **Naming Intuitiveness.** Clarity and descriptiveness of variable and test method names.
- **Code Layout.** Structure, logic, and formatting of the test code.
- **Assertion Quality.** Effectiveness and relevance of assertions for validating the focal method.
- **Adoption Efforts.** Ease of integrating the test case into real-world usage.

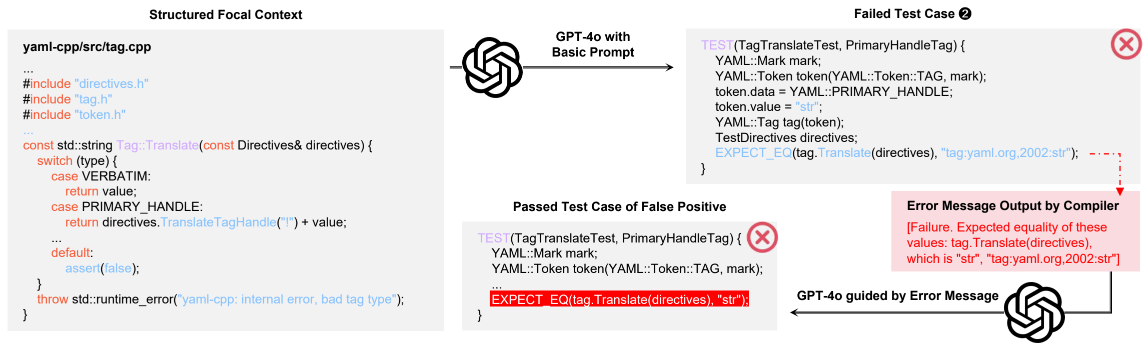
Table 20 summarizes the average scores across all participants. The results show that CITYWALK outperforms all LLM baselines in each of the four aspects. Specifically, CITYWALK surpasses the best baseline by 1.41% in **Naming Intuitiveness** (compared to DeepSeek-V3), 10.66% in **Code Layout** (compared to GPT-4o), 18.27% in **Assertion Quality** (compared to DeepSeek-V3), and 7.14% in **Adoption Efforts** (compared to DeepSeek-V3 and GPT-4o). Compared with human-written test cases, participants consider CITYWALK-generated test cases with more intuitive naming and more structured code. However, in terms of usability, CITYWALK-generated test cases demonstrate slightly lower assertion quality and comparable adoption effort. This is expected, as generating high-quality assertions remains an open challenge in LLM-based unit test generation. Overall, the findings highlight the superior readability and practical usability of CITYWALK-generated test cases, underscoring its value as a developer-assistive tool.

Table 20. Comparison of CITYWALK against the LLM Baselines in Terms of the Readability and Usability of Generated Test Cases

Approach	**Readability**		**Usability**	
	Naming Intuitiveness	**Code Layout**	**Assertion Quality**	**Adoption Efforts**
CodeGeeX4	2.70	2.32	1.46	2.47
DeepSeek-V3	2.83	2.39	2.08	2.52
GPT-3.5	2.55	2.26	1.75	2.38
GPT-4o	2.70	2.44	2.06	2.52
CITYWALK	2.87 (1.41% $\Uparrow$)	2.70 (10.66% $\Uparrow$)	2.46 (18.27% $\Uparrow$)	2.70 (7.14% $\Uparrow$)
Human-Written	2.64	2.63	2.87	2.75

5.4 False-Positive Executable Test Cases

Existing LLM-based unit test generation approaches [45, 51] primarily rely on compiler error messages to iteratively guide LLMs in fixing test cases that fail during execution. However, this strategy can lead LLMs to align expected values with observed outputs, resulting in test cases that pass without necessarily validating the intended functionality. As illustrated in Figure 12, when provided with the error message, GPT-4o directly uses the expected value "`str`" as the input for assertions, resulting in a passing test case that is, in reality, a false positive. While such behavior resembles


Structured Focal Context
GPT-4o with Basic Prompt
Failed Test Case ❷
Passed Test Case of False Positive
Error Message Output by Compiler
GPT-4o guided by Error Message

yaml-cpp/src/tag.cpp

```
...
#include "directives.h"
#include "tag.h"
#include "token.h"
...
const std::string Tag::Translate(const Directives& directives) {
  switch (type) {
    case VERBATIM:
      return value;
    case PRIMARY_HANDLE:
      return directives.TranslateTagHandle("!") + value;
    ...
    default:
      assert(false);
  }
  throw std::runtime_error("yaml-cpp: internal error, bad tag type");
}
```

```
TEST(TagTranslateTest, PrimaryHandleTag) {
  YAML::Mark mark;
  YAML::Token token(YAML::Token::TAG, mark);
  token.data = YAML::PRIMARY_HANDLE;
  token.value = "str";
  YAML::Tag tag(token);
  TestDirectives directives;
  EXPECT_EQ(tag.Translate(directives), "tag:yaml.org,2002:str");
}
```

[Failure. Expected equality of these values: tag.Translate(directives), which is "str", "tag:yaml.org,2002:str"]

```
TEST(TagTranslateTest, PrimaryHandleTag) {
  YAML::Mark mark;
  YAML::Token token(YAML::Token::TAG, mark);
  ...
  EXPECT_EQ(tag.Translate(directives), "str");
}
```

Fig. 12. False Positive Example for the Failed Test Case ❷ within Figure 1 using Iterative LLM-Based Fixing.

regression testing, a common practice in automated unit test generation, it risks reinforcing incorrect behavior when the underlying implementation contains faults.

Our goal is to highlight this specific risk introduced by compiler message–driven error-fixing strategies in LLM-based frameworks. When LLMs are exposed to runtime or compiler messages that reveal actual outputs, they may inadvertently "learn" to generate assertions that merely reproduce observed behavior rather than verify correctness. Thus, CITYWALK is intentionally designed to avoid relying on compiler feedback for fixing execution failures. Effectively detecting and addressing such false positives remains as our future work.

5.5 Threats to Validity

In this subsection, we discuss the primary threats to the validity of CITYWALK, as outlined below:

- **External Threat.** The primary threats to external validity lie in the diversity of the projects used for evaluation and its generalization to other LLMs. In this paper, we collect 1288 focal methods across ten real-world open-source C++ projects crawled from GitHub, ensuring a degree of diversity and quality in our evaluation. For comparison, we select two open-source code LLMs (CodeGeeX4 and DeepSeek-V3) and two closed-source commercial LLMs (GPT-3.5 and GPT-4o), considering their varied model sizes, architectures, and effectiveness on coding tasks. Future work will involve expanding the benchmark and integrating additional LLMs to better assess the generalizability of CITYWALK.
- **Internal Threat.** LLMs exhibit sensitivity to prompt configuration and hyper-parameter settings, particularly the number of task examples and the phrasing of natural language instructions, which can substantially influence performance. To ensure a fair comparison, we use consistent prompts and hyper-parameters across both CITYWALK and all baseline approaches. Additionally, we adopt empirically motivated default configurations rather than fine-tuning prompts or parameters through trial-and-error. We acknowledge that further improvements may be achievable through additional prompt and hyper-parameter tuning. Another potential threat to validity relates to data leakage issue. As GPT-4o is a closed-source model, the exact composition of its training data are not publicly disclosed. Despite this limitation, CITYWALK exhibits a significant improvement in generating high-quality test cases compared to GPT-4o, which utilizes the same underlying architecture. These enhancements suggest that the performance gains achieved by CITYWALK are not merely attributable to the model's memorization of its training data.

6 RELATED WORK

To mitigate the manual effort associated with writing unit tests for developers in practice, researchers have proposed various automation techniques aimed at enhancing testing efficiency. Existing approaches can be broadly categorized into the following three technical avenues.

6.1 Program Analysis-Based Automated Unit Testing Tools

EvoSuite [12] is an automated test case generation tool tailored for Java. It utilizes mutation testing and constraint-solving techniques to generate appropriate assertions, which effectively summarize the behavior of the program while maximizing the number of killed mutants. In contrast, Randoop [24, 25] is an automated tool that adopts feedback-driven random testing techniques to generate assertions. Randoop leverages the outcomes of test executions to generate assertions that accurately capture the program's behavior. Pynguin [18], an extendable test generation framework for Python to produce regression tests via search-based techniques. Coyote C++ [27] employs a sophisticated concolic execution-based method to facilitate fully automated unit testing for C and C++. Despite demonstrating commendable performance in achieving high code coverage, the aforementioned tools exhibit certain limitations: (1) The test code generated by existing tools often suffers from poor readability [7]; (2) The assertions produced by these tools are frequently insufficient in effectively detecting real-world faults [35, 36]; (3) Search-based techniques may encounter path explosion issues due to excessively large search spaces [41].

6.2 Pre-Trained Language Model-Based Automated Unit Test Generation

Tufano et al. [43] pre-trained a language model on large-scale unsupervised Java corpora, subsequently fine-tuning the model for unit test generation, thereby enabling the efficient generation of test cases. Zhang et al. [54] adopted the summarization of focal methods as complementary information to capture the developers' intent, which aids in generating meaningful test assertions for helping developers in writing accurate unit test cases. Similarly, Alagarsamy et al. [1] utilized both focal methods and assertion declarations during model pre-training, aiming to establish connections between focal methods and corresponding test cases. Shin et al. [37] developed project-specific datasets for domain adaptation by leveraging existing developer-written test cases within each project, promoting the generation of more human-like unit tests. Steenhoek et al. [40] employed reinforcement learning for model optimization, designing reward functions based on the static quality metrics of the generated unit test cases. Nonetheless, unit test cases generated through the paradigm of pre-training and fine-tuning frequently encounter compilation or execution failures [50].

6.3 LLM-based Automated Unit Test Generation

With the emergence of generative artificial intelligence, researchers have increasingly investigated LLM-driven approaches for automatically generating unit test cases [17, 44]. ChatUniTest [6] adaptively constructs dependency contexts for the focal method, and employs a generate-validate-fix mechanism to address errors in the generated test cases. Similarly, ChatTester [51] enhances the quality of generated test cases via intent comprehension and iterative correction. In contrast to prompt engineering-based approaches that merely append file-level dependencies as additional contextual information, CITYWALK conducts a comprehensive analysis of the project under test to extract project-level dependencies that impact test case generation, such as environment requirements. Moreover, CITYWALK employs RAG techniques to integrate language-specific knowledge from project documentation and source code, thereby enhancing performance. On the other hand, SymPrompt [30] guides LLMs to generate high-coverage test code by incorporating symbolic execution-based path information into the prompts. However, SymPrompt does not address the issue of path reachability, which can result in the inclusion of unreachable path information in the prompts, ultimately leading to inaccurate model reasoning. HITS [45] simplifies the analysis of complex focal methods through program decomposition and achieves high coverage scores by prompting LLMs to generate test cases for sliced code segments. Unlike previous approaches that primarily target interpreted languages such as Java and Python, CITYWALK is specifically designed to address C++-specific challenges. In this work, CITYWALK conducts an empirical analysis to identify common error patterns in LLM-generated C++ unit tests, It then documents these C++-specific failure patterns as insights to guide the post-processing of generated test cases, ensuring their accuracy and reliability.

7 CONCLUSION AND FUTURE WORK

This paper presents a novel framework CITYWALK designed to enhance the capabilities of LLMs in generating high-quality C++ unit test cases. We explore the potential of GPT-4o by integrating program analysis techniques with retrieval-augmented strategies, providing guidance through three key aspects: **project dependencies**, **intention contexts**, and **language-specific knowledge**. Additionally, we decompose the unit test generation task into three distinct stages, utilizing step-by-step instructions to streamline the generation of C++ test cases, complemented by effective post-processing techniques. Extensive experiments demonstrate the superiority of CITYWALK, and further ablation studies validate the contributions of each designed component within CITYWALK.

Future work will focus on improving LLM-generated unit test quality through two key advancements: (1) developing robust assertion verification techniques to validate functional correctness beyond coverage metrics, ensuring precise alignment with program specifications, and (2) enhancing bug-detection capabilities to identify diverse real-world software faults. These efforts aim to substantially narrow the gap between LLM-generated and human-written tests in terms of reliability and practical utility.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their insightful comments and suggestions. This work was partially supported by the National Key R&D Program of China (Grant No. 2024YFB4505902), the Major Project of ISCAS (Grant No. ISCAS-ZD-202302), the Basic Research Project of ISCAS (Grant No. ISCAS-JCZD-202403), the Youth Innovation Promotion Association of the Chinese Academy of Sciences (Grant Nos. Y2022044 and 2023121).

REFERENCES

[1] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2024. A3Test: Assertion-Augmented Automated Test Case Generation. *Inf. Softw. Technol.* 176 (2024), 107565. https://doi.org/10.1016/j.infsof.2024.107565

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

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

[4] Sushil Krishna Bajracharya, Joel Ossher, and Cristina Videira Lopes. 2010. Leveraging Usage Similarity for Effective Retrieval of Examples in Code Repositories. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, Santa Fe, NM, 157–166. https://doi.org/10.1145/1882291.1882316

[5] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking Large Language Models in Retrieval-Augmented Generation. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, Vancouver, 17754–17762. https://doi.org/10.1609/aaai.v38i16.29728

[6] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE)*. ACM, Porto de

Galinhas, 572–576. https://doi.org/10.1145/3663529.3663801

[7] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling Readability to Improve Unit Tests. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, Bergamo, 107–118. https://doi.org/10.1145/2786805.2786838

[8] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Naples, 201–211. https://doi.org/10.1109/ISSRE.2014.11

[9] Sourav Deb, Kush Jain, Rijnard van Tonder, Claire Le Goues, and Alex Groce. 2024. Syntax Is All You Need: A Universal-Language Approach to Mutant Generation. *Proc. ACM Softw. Eng.* 1, FSE (2024), 654–674. https://doi.org/10.1145/3643756

[10] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. *CoRR* abs/2412.19437 (2024). https://doi.org/10.48550/arXiv.2412.19437

[11] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss Library. *CoRR* abs/2401.08281 (2024). https://doi.org/10.48550/arXiv.2401.08281

[12] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, Szeged, 416–419. https://doi.org/10.1145/2025113.2025179

[13] Davide Fucci, Simone Romano, Maria Teresa Baldassarre, Danilo Caivano, Giuseppe Scanniello, Burak Turhan, and Natalia Juristo. 2018. A Longitudinal Cohort Study on the Retainment of Test-Driven Development. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, Oulu, 18:1–18:10. https://doi.org/10.1145/3239235.3240502

[14] Vahid Garousi and Junji Zhi. 2013. A Survey of Software Testing Practices in Canada. *J. Syst. Softw.* 86, 5 (2013), 1354–1376. https://doi.org/10.1016/j.jss.2012.12.051

[15] Robert Sebastian Herlim, Yunho Kim, and Moonzoo Kim. 2022. CITRUS: Automated Unit Testing Tool for Real-world C++ Programs. In *Proceedings of the 15th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Valencia, 400–410. https://doi.org/10.1109/ICST53961.2022.00046

[16] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[17] 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 IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, 2312–2323. https://doi.org/10.1109/ICSE48619.2023.00194

[18] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. In *Companion Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM/IEEE, Pittsburgh, PA, 168–172. https://doi.org/10.1145/3510454.3516829

[19] Walid Maalej and Martin P. Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Trans. Software Eng.* 39, 9 (2013), 1264–1282. https://doi.org/10.1109/TSE.2013.12

[20] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Trans. Software Eng.* 2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[21] OpenAI. 2022. *Introducing ChatGPT*. Technical Report. . https://openai.com/blog/chatgpt

[22] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/arXiv.2303.08774

[23] Lotfi Ben Othmane, Pelin Angin, Harold Weffers, and Bharat K. Bhargava. 2014. Extending the Agile Development Process to Develop Acceptably Secure Software. *IEEE Trans. Dependable Secur. Comput.* 11, 6 (2014), 497–509. https://doi.org/10.1109/TDSC.2014.2298011

[24] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Companion Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Montreal, Quebec, 815–816. https://doi.org/10.1145/1297846.1297902

[25] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Minneapolis, MN, 75–84. https://doi.org/10.1109/ICSE.2007.37

[26] Sanghoon Rho, Philipp Martens, Seungcheol Shin, and Yeoneo Kim. 2024. Taming the Beast: Fully Automated Unit Testing with Coyote C++. *CoRR* abs/2401.01073 (2024). https://doi.org/10.48550/arXiv.2401.01073

[27] Sanghoon Rho, Philipp Martens, Seungcheol Shin, Yeoneo Kim, Hoon Heo, and SeungHyun Oh. 2023. Coyote C++: An Industrial-Strength Fully Automated Unit Testing Tool. In *Joint Proceedings of the 5th International Workshop on Experience with SQuaRE series and its Future Direction and the 11th International Workshop on Quantitative Approaches to Software Quality co-located with the 30th Asia Pacific Software Engineering Conference (APSEC)*. CEUR-WS.org, Seoul, 45–50. https://ceur-ws.org/Vol-3612/QuASoQ_2023_Paper_01.pdf

[28] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). https://doi.org/10.48550/arXiv.2308.12950

[29] Per Runeson. 2006. A Survey of Unit Testing Practices. *IEEE Softw.* 23, 4 (2006), 22–29. https://doi.org/10.1109/MS.2006.91

[30] 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 (2024), 951–971. https://doi.org/10.1145/3643769

[31] David Saff and Michael D. Ernst. 2004. Mock Object Creation for Test Factoring. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE)*. ACM, Washington, DC, 49–51. https://doi.org/10.1145/996821.996838

[32] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. 2024. TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion. In *Companion Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. ACM, Lisbon, 30–34. https://doi.org/10.1145/3639478.3640024

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

[34] Sebastian Schweikl, Gordon Fraser, and Andrea Arcuri. 2022. EvoSuite at the SBST 2022 Tool Competition. In *Proceedings of the 15th IEEE/ACM International Workshop on Search-Based Software Testing (SBST@ICSE)*. IEEE, Pittsburgh, PA, 33–34. https://doi.org/10.1145/3526072.3527526

[35] Sina Shamshiri. 2015. Automated Unit Test Generation for Evolving Software. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, Bergamo, 1038–1041. https://doi.org/10.1145/2786805.2803196

[36] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Lincoln, NE, 201–211. https://doi.org/10.1109/ASE.2015.86

[37] Jiho Shin, Sepehr Hashtroudi, Hadi Hemmati, and Song Wang. 2024. Domain Adaptation for Code Model-Based Unit Test Case Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, Vienna, 1211–1222. https://doi.org/10.1145/3650212.3680354

[38] Davide Spadini, Mauricio Finavaro Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To Mock or Not to Mock?: An Empirical Study on Mocking Practices. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, Buenos Aires, 402–412. https://doi.org/10.1109/MSR.2017.61

[39] Davide Spadini, Maurício Finavaro Aniche, Magiel Bruntink, and Alberto Bacchelli. 2019. Mock Objects for Testing Java Systems - Why and How Developers Use Them, and How They Evolve. *Empir. Softw. Eng.* 24, 3 (2019), 1461–1498. https://doi.org/10.1007/s10664-018-9663-0

[40] Benjamin Steenhoek, Michele Tufano, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation. *CoRR* abs/2310.02368 (2023). https://doi.org/10.48550/arXiv.2310.02368

[41] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2024. ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. *IEEE Trans. Software Eng.* 50, 6 (2024), 1340–1359. https://doi.org/10.1109/TSE.2024.3382365

[42] Dave Thomas and Andy Hunt. 2002. Mock Objects. *IEEE Softw.* 19, 3 (2002), 22–24. https://doi.org/10.1109/MS.2002.1003449

[43] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit Test Case Generation with Transformers and Focal Context. *CoRR* abs/2009.05617 (2021). https://arxiv.org/abs/2009.05617

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

[45] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-Coverage LLM-Based Unit Test Generation via Method Slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, Sacramento, CA, 1258–1268. https://doi.org/10.1145/3691620.3695501

[46] Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. 2024. C-Pack: Packed Resources for General Chinese Embeddings. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, Washington, DC, 641–649. https://doi.org/10.1145/3626772.365787

[47] Tao Xie. 2006. Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*. Springer, Nantes, 380–403. https://doi.org/10.1007/11785477_23

[48] Shuhan Yan, Hang Yu, Yuting Chen, Beijun Shen, and Lingxiao Jiang. 2020. Are the Code Snippets What We Are Searching for? A Benchmark and an Empirical Study on Code Search with Natural-Language Queries. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, London, ON, 344–354. https://doi.org/10.1109/SANER48275.2020.9054840

[49] 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 (ASE)*. ACM, Sacramento, CA, 1607–1619. https://doi.org/10.1145/3691620.3695529

[50] Yanming Yang, Xin Xia, David Lo, and John C. Grundy. 2022. A Survey on Deep Learning for Software Engineering. *ACM Comput. Surv.* 54, 10s (2022), 206:1–206:73. https://doi.org/10.1145/3505243

[51] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1703–1726. https://doi.org/10.1145/3660783

[52] Shudan Zhang, Hanlin Zhao, Xiao Liu, Qinkai Zheng, Zehan Qi, Xiaotao Gu, Yuxiao Dong, and Jie Tang. 2024. NaturalCodeBench: Examining Coding Performance Mismatch on HumanEval and Natural User Queries. In *Findings of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, Bangkok, 7907–7928. https://doi.org/10.18653/v1/2024.findings-acl.471

[53] Yuwei Zhang. 2024. *Replicate Package of CITYWALK*. Zenodo. https://zenodo.org/records/14022506

[54] Yuwei Zhang, Zhi Jin, Zejun Wang, Ying Xing, and Ge Li. 2023. SAGA: Summarization-Guided Assert Statement Generation. *CoRR* abs/2305.14808 (2023). https://doi.org/10.48550/arXiv.2305.14808

[55] Ziyao Zhang, Yanli Wang, Chong Wang, Jiachi Chen, and Zibin Zheng. 2024. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. *CoRR* abs/2409.20550 (2024). https://doi.org/10.48550/arXiv.2409.20550

[56] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, Long Beach, CA, 5673–5684. https://doi.org/10.1145/3580305.3599790

[57] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro von Werra. 2024. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. *CoRR* abs/2406.15877 (2024). https://doi.org/10.48550/arXiv.2406.15877

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009