MR-Coupler: Automated Metamorphic Test Generation via Functional Coupling Analysis

ANONYMOUS AUTHOR(S)

Metamorphic testing (MT) is a widely recognized technique for alleviating the oracle problem in software testing. However, its adoption is hindered by the difficulty of constructing effective metamorphic relations (MRs), which often require domain-specific or hard-to-obtain knowledge. In this work, we propose a novel approach that leverages the functional coupling between methods, which is readily available in source code, to automatically construct MRs and generate metamorphic test cases (MTCs). Our technique, MR-Coupler, identifies functionally coupled method pairs, employs large language models to generate candidate MTCs, and validates them through test amplification and mutation analysis. In particular, we leverage three functional coupling patterns to avoid expensive enumeration of possible method pairs, and a novel validation mechanism to reduce false alarms. Our evaluation of MR-Coupler on 100 human-written MTCs and 50 real-world bugs shows that it generates valid MTCs for over 90% of tasks, improves valid MTC generation by 64.90%, and reduces false alarms by 36.56% compared to baselines. Furthermore, the MTCs generated by MR-Coupler detect 44% of the real bugs. Moreover, the code structures of these MTCs closely follow the human-written MR skeletons. Our results highlight the effectiveness of leveraging functional coupling for automated MR construction and the potential of MR-Coupler to facilitate the adoption of MT in practice. We also released the tool and experimental data to support future research.

ACM Reference Format:
Anonymous Author(s). 2025. MR-Coupler: Automated Metamorphic Test Generation via Functional Coupling Analysis. 1, 1 (September 2025), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Metamorphic testing (MT) is a powerful testing technique that helps address the test oracle problem [9, 33]. Specifically, MT tackles this challenge by leveraging relations between multiple outputs (i.e., output relation) given the relations between their corresponding inputs (i.e., input relation). In this case, the logical implication from input relation to output relation defines a *metamorphic relation (MR)*. MT has been shown to be effective in various software systems, especially where the expected outputs are difficult to specify (e.g., compilers [22, 36], machine translation [6, 46], question answering systems [45, 50], etc.). Moreover, one MR can serve as an oracle applied to many test inputs to exercise diverse program behaviors and enhance test adequacy [48].

Despite its potential, adoption of MT is challenging. A key bottleneck is the construction of effective MRs [33], which requires domain-specific knowledge. Although several attempts have been made to explore the generation of MRs, these approaches suffer from (i) reliance on manual effort [10, 37, 55], (ii) assumptions of regression testing scenarios [2, 3], (iii) restriction to specific domains (e.g., autonomous driving) [34, 52, 53, 55, 58], or (iv) requirements for high-quality specifications [5, 35, 48]. All these studies rely on knowledge that is hard to obtain. Although a recent study [48] reported the possibility of mining the fragmented knowledge required by MRs from test cases, it also found such test cases to be rarely available: they account for only 1% of the studied test cases

ACM XXXX-XXXX/2025/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and are scattered in only 20% of the studied projects. The lack of automatic methodologies for constructing metamorphic test cases (MTCs) hinders the widespread adoption of MT. To ease its adoption, a technique without the above-mentioned limitations is expected. To construct such a technique, *a central challenge is to formulate MRs without relying on knowledge that is hard to obtain.*

Fortunately, we found that the *functional coupling between methods, which is readily available in the code, can be formulated as MRs*. For example, the pair of functions `encrypt` and `decrypt` can formulate an MR $x = decrypt(encrypt(x))$, as shown in Listing 1. This motivates us to formulate MRs by identifying such coupled method pairs. This idea offers several advantages: (1) *readily-available knowledge*: it relies solely on a pair of methods and their implementation, which is by construction available in the scenario of unit testing, (2) *more tractable problem*: this transforms the challenging problem of deriving MRs into code understanding and relation reasoning, which can be effectively handled by current state-of-the-art large language models (LLMs) [24, 38, 47, 51]. For instance, although it is challenging to come up with MRs for a target method `encrypt`, when paired with a coupled method `decrypt`, it becomes easier for LLMs to understand their functionalities separately, realize that they are inverse functions, and then formulate a relation $x = decrypt(encrypt(x))$. (3) *easier bug manifestation*: certain bugs can be revealed more easily with coupled computations. For instance, while it is difficult to reveal the bug in Listing 2 by calling `encrypt` and `decrypt` separately, it becomes easier with the MR $x = decrypt(encrypt(x))$. In summary, leveraging functionally coupled methods as a foundation for MR construction provides a practical and effective pathway to automate metamorphic testing and broaden its applicability.

However, leveraging functionally coupled methods to construct MRs requires addressing two technical issues. First, given a target method, there can be dozens of candidate method pairs, and it is expensive to enumerate all possible method pairs blindly for MR construction. Thus, there is a need for a precise mechanism to identify functionally coupled method pairs, which provides better focal methods for subsequent MR construction. Second, while LLMs enable MTC generation via code understanding and reasoning, the resulting MTCs can be invalid due to hallucination [57]. Therefore, we need an effective mechanism to validate the generated MTCs, which allows us to avoid overwhelming developers with false alarms [40].

To tackle these technical issues and effectively generate MTCs, we propose MR-Coupler, an automatic MTC generator for a given target method. It operates in three phases. First, it identifies functionally coupled methods as ingredients for MR construction, based on their signatures and implementations. This addresses the first technical issue, based on our observation that developers often write MTCs for methods that operate on the same data structures or share common dependencies (e.g., APIs and class fields). Next, it employs LLMs to generate MTCs based on each identified functionally coupled method pair by providing relevant and minimal context. Specifically, we instruct LLMs to understand their functionalities and reason about potential MRs between them. To reduce hallucinations that lead to invalid code, we provide examples of MTCs and retrieve API usages for LLMs to follow. Finally, it validates the generated candidate MTCs via test amplification and mutation analysis. To validate the MTCs without a given ground truth, we create mutants from the original program by injecting artificial faults, and expect more amplified MTCs (from the candidate MTC) to pass on the original version compared with the faulty mutants. This filtering strategy is based on a property of MT: the MR embedded in a correct MTC should apply to many other inputs to effectively kill mutants [48].

We evaluated MR-Coupler on (i) 100 human-written MTCs with corresponding target methods and (ii) 50 real-world bugs as tasks for evaluation. MR-Coupler successfully generates valid MTCs for over 90% of tasks, achieving a 64.90% improvement in valid MTC generation and a 36.56% reduction in false alarms compared with baselines. The MTCs generated by MR-Coupler found 44% of the 50 real bugs. The key components play crucial roles in MR-Coupler: we found that

```java
@Test
public void testEncryptDecrypt() throws Exception {
    String plainText = "Hello AES!";
    SecretKey secKey = AESEncryption.getSecretEncryptionKey();
    // Encrypt the plaintext: invoke on the source input, and produce the source output
    byte[] cipherText = AESEncryption.encryptText(plainText, secKey);
    // Decrypt the ciphertext: invoke on the follow-up input, and produce the follow-up output
    String decryptedText = AESEncryption.decryptText(cipherText, secKey);
    assertEquals(plainText, decryptedText); // output relation assertion
}
```

Listing 1. An Example of MTC that Encodes the MR $x = decrypt(encrypt(x))$ over encryptText and decryptText

```java
public static byte[] encryptText(String plainText, SecretKey secKey) {
    Cipher aesCipher = Cipher.getInstance("AES");
    // aesCipher.init(Cipher.ENCRYPT_MODE, AESEncryption.defaultKey);   // bug
    aesCipher.init(Cipher.ENCRYPT_MODE, secKey);                         // fix
    return aesCipher.doFinal(plainText);
}
public static String decryptText(byte[] byteCipherText, SecretKey secKey) {
    Cipher aesCipher = Cipher.getInstance("AES");
    aesCipher.init(Cipher.DECRYPT_MODE, secKey);
    return aesCipher.doFinal(byteCipherText);
}
```

Listing 2. Code of Methods encryptText and decryptText

functional relevance between methods guides LLMs to produce valid, bug-revealing MTCs, while MTC amplification and validation steps further improve bug detection and halve false alarms. Last but not least, MR-Coupler can mimic developers in using functionally coupled methods and achieves over 90% MR-skeleton consistency with human-written MTCs, highlighting its potential to assist developers in constructing MTCs.

In summary, we make the following contributions in this paper.

- To the best of our knowledge, we are the first to construct MRs by leveraging the functional coupling between methods. Our approach relies solely on the code under test, and thus enables easier MR construction and lowers the barrier to adopting MT.
- We designed and implemented MR-Coupler, an automatic approach to generate concrete metamorphic test cases. MR-Coupler instructs LLMs with relevant and minimal context for MR construction, and validates the generated MTCs with a novel filtering mechanism based on mutation analysis.
- We conduct extensive experiments to evaluate the effectiveness of MR-Coupler, including the validity of generated MTCs, the capability of revealing real bugs, and the similarity of generated MTCs to human-written MTCs.
- We made MR-Coupler and our experimental data publicly available to facilitate future research. Our artifact is available at the website of MR-Coupler [42].

2 PRELIMINARIES

Metamorphic Testing (MT) evaluates a program P using a Metamorphic Relation (MR, $\mathcal{R}$), which is a logical implication from an input relation $\mathcal{R}_i$ to an output relation $\mathcal{R}_o$ [9, 33, 48].

$$\mathcal{R} : \mathcal{R}_i \left(x_s, x_f\right) \implies \mathcal{R}_o \left(x_s, x_f, y_s, y_f\right)$$

$\mathcal{R}_i$ specifies the rule for deriving a *follow-up input* (x_f) from a given *source input* (x_s), while $\mathcal{R}_o$ defines the expected relationship over the inputs and their corresponding outputs (x_s, x_f, y_s, y_f)[1].

A *Metamorphic Test Case (MTC)* is a test case that implements MT. As illustrated in Listing 1, given an MR $\mathcal{R}$ ($x = decrypt(encrypt(x))$) for a target program P (the class `AESEncryption`), an MTC consists of the following steps: (i) constructing a source input x_s (`plainText, secKey`), (ii) executing P (`AESEncryption.encryptText`) on x_s to obtain the source output y_s (`cipherText`), (iii) constructing a follow-up input x_f that satisfies $\mathcal{R}_i$ (`cipherText` also serves as the follow-up input), (iv) executing P (`AESEncryption.decryptText`) on x_f to obtain the follow-up output y_f (`decryptedText`), and (v) verifying if the output relation $\mathcal{R}_o$ is satisfied (`assertEquals(plainText, decryptedText)`).

Note that, following the definition by Xu et al. [48], the class `AESEncryption` is the *target program* P under test, and `encryptText` and `decryptText` are the *target methods* – the specific components of P exercised during MT execution. The methods `encryptText` and `decryptText` are functionally coupled and form a method pair for formulating the MR $x = decrypt(encrypt(x))$.

Unique advantages of MT. MT offers several advantages: (i) *Detecting faults in "non-testable" programs.* MT is particularly valuable for validating *non-testable programs* whose expected outputs for given inputs are difficult or infeasible to specify [9, 33]. Numerous empirical studies have demonstrated the effectiveness of MT in revealing faults in such scenarios [8, 25, 27, 34, 47, 48]. (ii) *Reusable oracle across inputs.* A key advantage of MT is that a single MR can be applied to many inputs, enabling the test suite to exercise a broader range of program behaviors and thereby improve its fault-detection capability. For example, in the AES case (Listing 1), the MR $x = decrypt(encrypt(x))$ can be applied not only to an original input such as `"Hello AES!"`, but also to corner cases such as an empty string (`plainText=""`) or strings containing special characters (`plainText="~!@#$%^&*()_+"`).

Recent studies [47, 48] have further highlighted the practical benefits of MTCs. Some open-source projects contain human-written MTCs, and these MTCs can improve test adequacy by generalizing their MRs to new inputs. However, it was found that only about 1% of test cases in the studied projects are MTCs, and 80% of the projects contain none at all [48]. This indicates that, for most programs under test, valuable MRs and the corresponding MTCs are still missing.

Goal. This motivates our work: to develop a *fully automated, domain-agnostic approach* to generate MTCs directly from code. Our approach aims to bridge this gap by leveraging functional coupling between methods as the basis for formulating MRs, and by automatically generating valid MTCs that can be applied to diverse inputs to enhance test adequacy.

3 APPROACH: MR-COUPLER

In this section, we present MR-Coupler, an automated MTC generator based on functional coupling. Figure 1 presents an overview of MR-Coupler. Given a target method and its container class as input, MR-Coupler produces a set of MTCs. Specifically, the generation process consists of three phases:

(1) *Coupled Methods Identification.* In the first phase, MR-Coupler identifies functionally coupled methods that will be paired with the target method for MR construction. These methods are relevant to the target method in their intention or functional behavior. Such relevance can lead to potential MRs over their functionalities.
(2) *MTC Generation.* In the second phase, MR-Coupler leverages LLMs to generate MTCs for the method pairs yielded by the first phase. To mitigate the impact of LLM hallucination [57], we

[1]We incorporate both inputs and outputs in the output relation to present a more general definition of output relation that fits the MRs asserting the output relation based on inputs, such as round-trip translation [48].

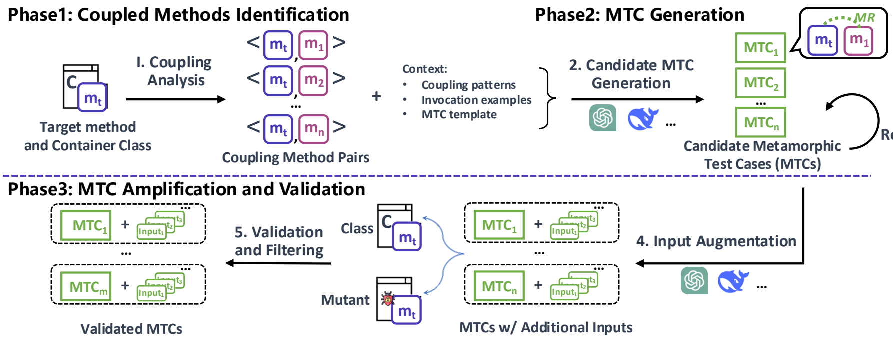
Phase1: Coupled Methods Identification
Target method and Container Class
I. Coupling Analysis
Coupling Method Pairs
Context:
Coupling patterns
Invocation examples
MTC template
Phase2: MTC Generation
2. Candidate MTC Generation
Candidate Metamorphic Test Cases (MTCs)
3. MTC Refinement
MR
Phase3: MTC Amplification and Validation
4. Input Augmentation
MTCs w/ Additional Inputs
Class
Mutant
5. Validation and Filtering
Validated MTCs

Fig. 1. An overview of MR-Coupler

provide example usage of the involved methods and a template of MTC in the prompt. We also perform subsequent refinement based on the execution output of the generated MTCs.

(3) *MTC Amplification and Validation.* The third phase amplifies the MTC generated by the second phase with additional inputs. This phase serves two goals. First, this serves as a validation for the MTC based on a necessary property: a valid MTC should not have a lower pass rate on the original version than on the mutants. Second, the amplified MTCs can help reveal more bugs by exercising a broader range of program behaviors.

3.1 Phase 1: Coupled Methods Identification

This phase is to identify the functionally coupled methods to combine with the target method as method pairs for MR construction. However, given a target method and its container class, it is non-trivial to identify the functionally coupled methods that are suitable for MR construction.

On the one hand, classes often contain dozens of methods. For example, in our dataset, there are over 30 candidate methods in a class for each target method on average. Naively including all candidates not only increases the cost of LLM tokens but also risks of introducing noisy context, which distracts LLMs. The difficulty is to identify those methods with relevance (e.g., producer–consumer, equivalent or inverse functions) suitable for MR construction.

3.1.1 Characterization of Method Coupling. Although the number of possible method combinations is large, we found the method pairs used for MR construction often exhibit certain patterns in their relation: sharing relevant intentions, similar implementation behavior, or state interference.

For example, for two methods that have the inverse relationship, methods perform opposite input and output type transformations (e.g., `encryptText:(String,SecretKey)->byte[]` and `decryptText:(byte[],SecretKey)->String`) [41]. For two methods that have the producer-consumer relationship, one method updates or produces states (e.g., fields of an object), and another accesses them [32]. For two methods that have an equivalent or similar functionality relationship, e.g., overloading methods, they invoke the same APIs, access or update the same fields of an object. For two methods that have the composition or specialization relationship, one method internally calls or extends the functionality of another [14]. More examples can be found in our artifact [42].

Despite the diversity in these relationships, we found that most of them can be captured by three complementary patterns:

(1) *Relevant Intention*: The method pairs that are designed to perform related functionalities. This can be captured from their signatures, including method name, parameter types, and return type, which indicate the data types they consume and produce.

(2) *Similar Implementation Behavior*: The method pairs that have similar behaviors in their implementations. This can be captured by analyzing their function calls, indicating that they perform comparable or related operations.
(3) *Potential State Interference*: The method pairs where the invocation of one method can affect the behaviors of the other. This can be reflected by the class fields and object states that they access or modify.

3.1.2 Coupling Analysis. The step aims to identify functionally coupled methods characterized by the above three patterns.

Relevant Intention. Methods with relevant intention can often lead to potential MRs (e.g., methods `encryptText` and `decryptText`). To identify such method pairs, we analyze the method name, parameters, and return types. Specifically, given a target method m_t, a candidate method m_i is considered to have relevant intention if: (i) m_t and m_i share the same method name (indicating an overloading relationship of similar purpose), or (ii) m_t and m_i share common name tokens and operate on the same parameter or return types, which increases the likelihood that they manipulate the same data structures for relevant functionalities. These heuristics allow MR-Coupler to identify methods with relevant intentions (e.g., inverse methods, overloading methods, etc.).

Similar Implementation Behavior. Methods with relevant functionalities may not always come with similar names or signatures. Their relevance can manifest at the behavioral level. Therefore, to capture such relevance, MR-Coupler extracts the set of functions (e.g., APIs) invoked within each method. It then analyzes three types of invocation relationships: (i) whether m_t directly invokes m_i (or vice versa), indicating relationships like specialization or composition, (ii) whether m_t and m_i share common invoked APIs, suggesting similar behavior. This allows MR-Coupler to identify methods with similar implementation behavior.

Potential State Interference. Beyond invocation and API usage, method pairs can lead to potential MRs if one method can affect the behavior of the other. Such a relationship is often reflected by the field access and updates. Therefore, MR-Coupler analyzes the fields accessed and updated by each method. Given a target method m_t, it considers m_i relevant if: (i) m_i updates fields that m_t later reads (or vice versa), indicating a data-flow dependency and relationship like producer-consumer, (ii) m_i and m_t access the same fields, or (iii) m_i and m_t update the same fields, suggesting similar behavior. This allows MR-Coupler to identify potential state interference between methods.

Given a target method, MR-Coupler analyzes all the methods within its container class, and yields the method pairs that match any of the patterns. Note that a method pair may satisfy multiple patterns.

3.2 Phase 2: MTC Generation

Given the set of coupling method pairs, this phase employs LLMs to come up with MRs and generate concrete MTCs that conduct MT. However, generating valid MTCs remains challenging even with state-of-the-art LLMs [54, 56]. Many methods require complex object instantiations, parameter configurations, or specific environmental setups, making valid method invocation difficult. Without concrete usage examples, LLMs are prone to hallucinations, often producing code that references non-existent classes, APIs, or fields. Moreover, generated test cases must conform to the steps of MT (i.e., constructing source and follow-up inputs, invoking methods, and asserting output relation)

To address these challenges, MR-Coupler firstly provides LLMs with contextual guidance (e.g., method invocation examples and MTC template). After prompting the LLMs to generate MTCs, MR-Coupler further refines them based on the execution output.

~~~
You are an expert in Java programming and metamorphic testing, your task is to: ...

# Code of the paired method
    ```java
            byte[] encryptText(String plainText, SecretKey secKey) {
                    ...
            String decryptText(byte[] byteCipherText, SecretKey secKey) {
                    ...
    ~~~

# Coupling patterns on the paired methods
### Relevant Intention:
    * `encryptText` and `decryptText` operate on the same set of parameters and return types,
       but with different transformations.
        * `encryptText`: (String, SecretKey) -> byte[] , `decryptText`: (byte[],SecretKey) -> String
### Similar Implementation Behavior:
    * both `encryptText` and `decryptText` invoke same APIs: `Cipher.getInstance(``AES'')`
        ...
# Invocation examples
        ...
# Skeleton of the container class
        ...
# Deliverable
    ```java
    public class $testClass${
        @Test
        public void $testCase$() {
            <MTC Template>
        ...
    ```
```

Listing 3. Prompt Template for MTC Generation

3.2.1 *Candidate MTC Generation.*

Invocation example preparation. To help LLMs construct valid method invocations, MR-Coupler retrieves method invocation examples from the project under test and uses them as part of the contextual guidance.

Specifically, for each method pair $\langle m_t, m_i \rangle$, MR-Coupler searches for invocations of any of the two methods in the test code. MR-Coupler scans the test files (under the `/test/` directory) in the project under test and uses `JavaParser` [19] to identify test methods annotated with `@Test`. It then checks whether each test invokes either m_t or m_i. If so, the test is collected as an invocation example. All retrieved examples are aggregated, with at most three examples retained for each method, and provided to the LLMs as contextual guidance, increasing the likelihood that generated tests correctly instantiate objects and invoke methods.

Prompt Design. Listing 3 shows a simplified prompt template used by MR-Coupler for MTC generation. Referring to the prompt design in recent studies [47, 49, 51], the prompt includes: (i) a system message specifying the role of LLM and its tasks, (ii) the code of the paired methods, (iii) the identified coupling patterns on the paired method (iv) invocation examples, (v) the skeleton of the container class (fields and method signatures), and (vi) a MTC template that specifies the required deliverable. This structured prompt provides both contextual information (ii–v) and task description (i and vi), guiding the LLM to generate syntactically correct MTC. The details of employed LLMs and their configuration can be found in Section 4.2.1. The output of this step is a set of candidate MTCs, each implemented as a standalone test class.

3.2.2 *MTC Refinement.*

Consistent with prior observations [15, 47], LLM-generated code frequently fails to execute commonly due to errors such as “`cannot find symbol`”. These errors typically arise from two sources: (i) referencing non-existent classes, APIs, or fields (hallucinations), or (ii) missing dependencies (e.g., absent import statements).
```

~~~
```java
public class AESEncryptionTest{
    @Test
    public void testEncryptDecrypt_input1() {
        String text = "Hello!"; SecretKey key = AESEncryption.getSecretEncryptionKey();
        byte[] encryptedText = AESEncryption.encryptText(text, key);
        String decryptedText = AESEncryption.decryptText(encryptedText, key);
        assertEquals(text, decryptedText);
    }
    @Test
    public void testEncryptDecrypt_input2() {
        String text = NULL; SecretKey key = NULL;
        ...
    public void testEncryptDecrypt_input3() {
        String text = "~!@"; SecretKey key = AESEncryption.getSecretEncryptionKey();
        ...
    public void testEncryptDecrypt_input4() {
        String text = "_1234567890"; SecretKey key = AESEncryption.getSecretEncryptionKey();
        ...
    public void testEncryptDecrypt_inputM() {
        String text = ""; SecretKey key = AESEncryption.defaultKey;
        ...
}
```
~~~

Listing 4. Simplified example of an MTC with M additional inputs

To deal with this issue, MR-Coupler refines each non-compilable or non-executable MTC. First, the error message is provided back to the LLM to request an automatically revised version. If the revised MTC still fails to execute, MR-Coupler tries to fix the missing dependencies issue by statically analyzing the code using `JavaParser` to extract unresolved class names and searches for potential classes defined or imported in the project under test, and then adds the necessary import statements. Finally, this phase results in a refined set of MTCs, which are subsequently used for amplification and validation.

3.3 Phase 3: MTC Amplification and Validation

This phase aims to validate the candidate MTC generated in the previous phase and diversify the test inputs. Specifically, we amplify the MTC with additional inputs, and leverage a property of MT to refute invalid MTCs. This is because the previous phase can generate MTCs that encode *invalid* metamorphic relations (MRs). To refute such MTCs, we opted for mutation analysis: we create mutants from the original program by injecting artificial faults, and expect more amplified MTCs (from the candidate MTC) to pass on the original version compared with the faulty mutants. This filtering strategy is based on a property of MT: the MR embedded in a correct MTC should apply to many other inputs to effectively kill mutants [48]. In addition, as a side product, the amplified MTCs can also help exercise a broader range of program behaviors and increase their bug-revealing capability.

3.3.1 *Input Augmentation.*

MR-Coupler amplifies each MTC by generating additional inputs to its MR to exercise a broader range of program behaviors, thereby enhancing its bug-revealing capability. To generate these inputs, MR-Coupler employs LLMs by appending new instructions to the conversation for MTC generation and prompts the model to: (i) review the previous conversation and context, (ii) review the previously generated MTC, (iii) apply its MR to new inputs by replacing the original input with ⟨M⟩ new inputs (such as boundary values, random data, or special characters)

```java
public static byte[] encryptTextWithAbecedarium(String plainText,SecretKey secKey,String
↪ abecedarium)
{
    Cipher aesCipher = Cipher.getInstance("AES");
    aesCipher.abecedarium = AESEncryption.abecedarium;              // bug
    // aesCipher.abecedarium = abecedarium;                        // fix
    aesCipher.init(Cipher.ENCRYPT_MODE, secKey);
    return aesCipher.doFinal(plainText);
}
```

Listing 5. Example of `encryptText` and `encryptTextWithAbecedarium` with bugs

in the form of new test cases ($M = 10$ by default), and (iv) output the new test cases within the same class following the naming convention (`testMTC_newInput1()`, . . . , `testMTC_newInputM()`). Listing 4 shows a simplified example of amplified MTC with 5 new inputs.

3.3.2 Validation and Filtering. Given an amplified MTC with additional inputs, MR-Coupler executes it on both the *original* version of the target program, and a *mutated* version produced by injecting faults using Major [28].

For each MTC, MR-Coupler computes the pass rate p on the original version and p' on the mutated version. The validation property requires that $p > p'$: a valid MTC should pass consistently on a correct implementation and fail on a buggy implementation. If this property is violated ($p < p'$), the MTC is flagged as invalid and discarded. For instance, consider a mutant (`encryptText` uses wrong `key` shown in Listing 2) and a valid MR $decrypt(encrypt(x)) = x$ with additional inputs in Listing 4. Most inputs pass on the original version ($p = 80\%$, except when `text=NULL` throws an illegal input exception) but fail on the mutant ($p' = 20\%$, since only the case where `key = AESEncryption.defaultKey` coincidentally matches the default key succeeds). Because $p > p'$, this MTC is retained as valid. By contrast, we observed an LLM-generated invalid MRs $encrypt(plainText, secKey) = encryptTextWithAbecedarium(plainText, secKey, abecedarium)$. When tested against a mutant where `encryptTextWithAbecedarium` fails to set up the `abecedarium` (Listing 5), most inputs in Listing 4 fail on the original version ($p = 20\%$, except when the user-defined `abecedarium` happens to match the default). On the mutant, all inputs pass ($p' = 100\%$) because the `abecedarium` is ignored entirely. Since $p < p'$, this MTC is classified as invalid and filtered out.

When $p = p' = 100\%$, two interpretations are possible: (i) the injected mutants are ineffective and do not affect the tested behavior, or (ii) the MTC is ineffective in exposing mutants. In such cases, MR-Coupler conservatively retains the MTC. Since $p \geq p'$ is a necessary condition for a valid oracle, increasing the number of additional inputs raises the likelihood of exercising diverse program behaviors and triggering differences between the original and mutated versions, thereby improving the effectiveness of MR-Coupler's validation.

After validating each generated MTC, MR-Coupler finally outputs validated MTCs.

4 EVALUATION

In this section, we present our evaluation of MR-Coupler. Specifically, we aim to answer the following research questions (RQs).

RQ1 Validity: *How effective is MR-Coupler at generating MTCs?* This RQ investigates the overall effectiveness of MR-Coupler. Specifically, we assess MR-Coupler regarding the validity of the generated MTCs, i.e., whether they are syntactically correct, entail necessary steps of MT, and do not produce false alarms. In addition, we compared MR-Coupler with vanilla-LLM-based baselines to understand the superiority of our approach.

- **RQ2 Bug-Revealing Capability:** *How effective is MR-Coupler in revealing real-word bugs that discovered by human-written MTCs?* This RQ aims to understand the effectiveness of MR-Coupler in revealing bugs in practical scenarios. Compared to seeded bugs (e.g., mutants), real-world bugs are often more sophisticated. Thus, we evaluate whether the MTCs generated by MR-Coupler can detect real-world bugs as the human-written MTCs do in the same test subjects.
- **RQ3 Abalation Study:** *How does each step contribute to the effectiveness of MR-Coupler?* MR-Coupler incorporates three key steps: *coupling analysis* to identify functionally related methods, *input augmentation* to amplify generated MTCs, and *mutation analysis* to validate MTCs. This RQ performs an ablation study to understand how each of these steps contributes to the overall effectiveness of MR-Coupler in generating valid and bug-revealing MTCs.
- **RQ4 Similarity:** *Do the MTCs generated by MR-Coupler share the same MR skeletons as human-written ones?* This RQ evaluates whether the MTCs generated by MR-Coupler can mimic developers' practices in selecting functionally coupled method pairs and constructing input and output relations. This demonstrates the potential of MR-Coupler to assist developers in MTC construction, facilitating developers in integrating the generated tests into their codebase and easing subsequent maintenance.

4.1 Datasets

We prepared two datasets to answer the four RQs. The first dataset includes pairs of target methods together with corresponding human-written MTCs available in open-source projects to evaluate the validity and similarity of the generated MTCs (RQ1 and RQ4). The other is a subset from the first dataset, including only the cases whose MTCs can reveal bugs on a historical buggy version of the target program, for evaluating the bug-revealing capability of generated MTCs (RQ2).

4.1.1 Human-Written MTCs. The first dataset contains 1,471 MTCs written by developers in open-source Java projects. Each entry in this dataset consists of a human-written MTC and a corresponding pair of MR-coupled methods. These MTCs are valid and executable. Such a dataset is leveraged to (i) evaluate the validity of automatically generated MTCs (RQ1), by running them on executable target methods, and (ii) measure the similarity between the human-written MTCs and MR-Coupler generated MTCs, by checking whether they encode the same MR-skeletons (RQ4).

To construct such a dataset, we adopted a strategy similar to Xu et al. [48]. Specifically, we collected a list of high-quality Java projects (i.e., with at least 50 stars) from GitHub. The query was done on December-16, 2024, which returned over 24,000 projects. We then ran MR-Scout [48] to discover human-written MTCs from these projects, which yielded 46,006 candidate MTCs. With these candidates, we applied three filtering criteria to select the valid and executable MTCs:

(i) they must compile, as we need to compile and run the test cases in our experiments;
(ii) they must pass in the latest version of the project to ensure the MTCs are valid; and
(iii) the commit introducing these MTCs must mention an issue number in its commit message (e.g., containing "`#123`"). We prioritize such tests since they are often extensively discussed and reviewed to disclose the issues and thus tend to be of high quality.

The whole processes yielded a dataset containing 1,471 entries. Each entry in this dataset consists of a human-written MTC and a corresponding pair of MR-coupled methods. We ran MR-Scout [48] to obtain the corresponding pair of MR-coupled methods for each MTC. For example, in the MTC that encodes the relation $x = decrypt(encrypt(x))$ (Listing 1), `encryptText` and `decryptText` are the MR-coupled methods and will be identified by MR-Scout. We use the first invoked method `encryptText` as the target method and take `decryptText` as the ground truth of a coupled method. Each entry formulates an MTC generation task used for our experiments of RQ1 and RQ4.

4.1.2 Bug-revealing MTCs. The other dataset is made up of 50 entities with bug-revealing MTCs filtered from the first dataset. These entities are used to evaluate whether MR-Coupler can generate effective MTCs to reveal real bugs as the human-written MTCs do (RQ2). We identify such entities from the first dataset by checking whether their MTC will fail on a buggy version while pass on a fixed version. Specifically, an issue report may be resolved through commits; thus, for each issue-associated MTC, we identify two versions of the project: (i) the potential *buggy version*, defined as the commit before all issue-related commits; (ii) the potential *fixed version*, defined as the last commit of all issue-related commits. We then execute the MTC on both versions. We consider an MTC bug-revealing only if it fails on the buggy version and passes on the fixed version.

This process required significant manual effort to set up specific project environments for multiple versions of each project and resolve complicated dependency issues. We ultimately reproduced 50 MTC-bug pairs, obtaining their corresponding buggy and fixed program versions, which form the benchmark for evaluating the bug-revealing capability of MR-Coupler (RQ2).

4.2 Evaluation Setup

In this section, we present our evaluation setup. We introduce the LLMs we used, baselines, and the experiment environment.

4.2.1 Employed Large Language Models. MR-Coupler employs LLMs to generate MTCs and their alternative inputs. In the evaluation, we include representative state-of-the-art LLMs [16], covering general-purpose, coding, and reasoning LLMs from well-known model families. Specifically, they are *GPT-4o mini* from OpenAI [30],*Qwen3-coder-Flash* from Alibaba [1], *DeepSeek-V3.1* and *DeepSeek-V3.1-Think* from DeepSeek [11]. Following a typical setup in recent studies [7, 13, 47], for each MTC generation task, we repeated the generation process five times with a temperature setting of 0.2.

4.2.2 Baselines. To the best of our knowledge, there is no existing fully automated and domain-agnostic approach to generate metamorphic test cases for a given program under test. Although some approaches are proposed to generate domain-specific MRs [52, 55], or synthesize MRs based on human-prepared materials [29, 47, 48] or manual effort [35] (discussed in Section 5), adapting them into comparable automated domain-agnostic baselines is non-trivial. Given the proven effectiveness of LLMs in code [23, 24, 47] and test generation [38, 51], we set *directly prompting LLMs* as a baseline. In this baseline, we allow LLMs to conduct a round of revision to the generated code based on the execution feedback as in our method, which is found to be an effective common post-processing to enhance code generation [51]. The baseline uses a similar prompt template (Listing 3), and follows the same refinement step in Section 3.2.2.

Experimental results show that a target method can be paired with 6.93 relevant methods (rounded to 7) by MR-Coupler on average. Therefore, to have a fair comparison, we instructed the baseline LLMs to generate 8 candidate MTCs, which correspond to the target method itself, plus the 7 additional relevant methods. For each task, we repeat the generation process five times with a temperature setting of 0.2, consistent with MR-Coupler's configuration.

4.2.3 Experimental Environment. All experiments were conducted on a machine with a 64-core AMD Ryzen Threadripper PRO 3995WX CPU and 512 GB RAM. The LLMs in our evaluation are running on cloud platforms and accessed via the official APIs of OpenAI, Alibaba, and DeepSeek.

4.3 RQ1: Validity of Generated MTCs

RQ1 aims to evaluate the overall effectiveness of MR-Coupler in generating valid MTCs. To this end, we run MR-Coupler on the target methods in the dataset of human-written MTCs. We also compare it against the baselines (Section 4.2).

Table 1. Effectiveness of MR-Coupler in Generating Valid MTCs for 100 Target Methods

Metric	GPT-4o-mini		Qwen3-coder-Flash		Deepseek-V3.1		Deepseek-V3.1-Think		Improv.
	Baseline	MR-Coupler	Baseline	MR-Coupler	Baseline	MR-Coupler	Baseline	MR-Coupler	
Num. of Generated TCs	3923	4176	3984	3911	3968	3626	3971	4151	-
Pct. of Executable MTC	50.40%	83.69%	60.02%	92.66%	60.26%	93.08%	62.08%	88.44%	**54.35%**
Pct. of Valid MTC	40.66%	71.84%	47.52%	80.98%	52.60%	84.94%	55.35%	83.57%	**64.90%**
Num. of Successful Tasks	62	92	76	91	82	95	85	98	**24.82%**
Pct. of False Alarm	19.32%	14.16%	20.70%	12.61%	12.71%	8.74%	10.83%	5.50%	**36.56%**

4.3.1 Experiment Setup. Experiments were conducted at scale using four LLMs, with each MTC generation task repeated five times (Section 4.2.1). Running MR-Coupler and the baselines on all 1,471 tasks in the human-written MTC dataset is time-consuming and unaffordable. Therefore, we run MR-Coupler and baseline approaches on 100 randomly sampled entries in the dataset. Such a sample size ensures a confidence level of 95% and a margin of error $\leq$ 10%.

For each task (i.e., target method), MR-Coupler generates multiple MTCs. We present the total number of generated test cases, and measure the effectiveness using the following metrics:

- *Percentage of Executable MTC:* The proportion of executable MTCs to all generated test cases, where an MTC is executable if it (i) compiles and runs without errors, (ii) satisfies the necessary properties of an MTC. Following the definition from Xu et al. [48], an MTC must meet two properties: (*P1*) it must contain at least two method invocations with two inputs separately, and (*P2*) it must contain one assertion checking the relation between the inputs and outputs of the method invocations in *P1*. We re-ran their tool to automatically verify each generated test case.
- *Percentage of Valid MTC:* The proportion of valid MTCs to all generated test cases, where a valid MTC is an executable MTC that passes on the latest project version. We assume the latest version of a target method is of low probability to be buggy, as it has passed human-written MTCs.
- *Number of Successful Tasks:* The number of target methods for which at least one valid MTC is generated.
- *Percentage of False Alarm:* The proportion of invalid MTCs to all executable MTCs, where an invalid MTC satisfies the properties but fails on the latest version.

4.3.2 Experiment Results. Table 1 shows the result of MR-Coupler in generating valid MTCs for 100 target methods. MR-Coupler successfully generated valid MTCs for over 90 target methods, with its best performance achieved when using *DeepSeek-V3.1-Think*. Specifically, with *DeepSeek-V3.1-Think*, MR-Coupler produced valid MTCs for 98 of 100 target methods, with only 5.5% of false alarms. Compared to the baseline, MR-Couplerachieves a 64.90% higher valid MTC percentage with a 36.56% fewer false alarms. Even with the weakest model (*GPT-4o mini*), MR-Coupler still outperformed a baseline with a much more powerful model (i.e., *DeepSeek-V3.1-Think*) in generating valid MTCs.

The improvement in valid MTC generation can be attributed to two key factors. On the one hand, providing LLMs with functionally coupled methods serves as a hint, effectively inspiring them to infer valid MRs and then generate valid MTCs, thereby reducing hallucinations. Without such context, coming up with MRs from scratch is challenging for LLMs, leading to higher rates of invalid MRs. On average, MR-Coupler identified 6.93 relevant methods per target method from 30.44 candidate methods in their container classes, significantly narrowing the enumeration space. On the other hand, retrieving real invocation examples helps LLMs construct valid input objects and correctly invoke methods, particularly when methods require complex object instantiations.

For example, the target method `estimateCNF` [39] requires a less common `CNFEstimation` object as input. Without a concrete example, LLMs frequently failed to construct the object correctly and

Table 2. Bug-Revealing Results of MR-Coupler on 50 Bugs

Metric	GPT-4o-mini		Qwen3-coder-Flash		Deepseek-V3.1		Deepseek-V3.1-Think		Improv.
	Baseline	MR-Coupler	Baseline	MR-Coupler	Baseline	MR-Coupler	Baseline	MR-Coupler	
Num. of Generated TCs	1987	2740	1976	2086	2024	1797	2007	2661	-
Pct. of Bug-revealing MTCs	3.84%	6.53%	4.14%	7.29%	5.21%	6.60%	3.92%	7.77%	**67.65%**
Num. of Revealed Bugs	4	15	5	20	7	16	7	22	**229.46%**

even hallucinated non-existent APIs such as `getEstimator()`. This context helps generate 83.69% to 93.08% executable MTCs, which are 54.35% more compared with baselines.

Failure analysis. Even built with *DeepSeek-V3.1-Think*, MR-Coupler fails to generate any valid MTCs for two target methods and still exhibits a 5.5% false-alarm rate. The main reasons are as follows: (i) Some target methods require access to external or environmental resources (e.g., a JSON file or environment variable), but no invocation examples were available in the repository as the context. As a result, MR-Coupler fails to configure these resources in the generated MTCs, leading to non-executable tests. (ii) MR-Coupler relies on Major [28] to generate mutants for validation (Section 3.3.2). For some target methods, Major failed to execute due to environmental issues (e.g., uncompilable dependencies), preventing mutant generation and disabling the validation step that filters false alarms. (iii) In some cases, the generated MTCs cannot reveal the injected mutants, and do not expose behavioral differences between the base and mutated versions – the pass rates are identical, causing MR-Coupler to retain false alarms.

Answer to RQ1: MR-Coupler successfully generates valid MTCs for over 90% of tasks with, achieving 64.90% and 36.56% improvements in generating valid MTCs and reducing false alarms, respectively, compared with baselines.

4.4 RQ2: Bug-revealing capability

4.4.1 Experiment Setup. This RQ evaluates the capability of MR-Coupler in revealing real bugs, especially for those originally discovered by metamorphic test cases. With the collected 50 MTC-bug pairs (Section 4.1), for each bug, we take the buggy method as the target method. We measure the bug-revealing capability by the following two metrics:

- *Percentage of Bug-Revealing MTCs:* The proportion of generated MTCs that are bug-revealing, where a bug-revealing MTC is defined as an executable MTC that fails on the buggy version but passes on the fixed version of the target method.
- *Number of Revealed Bugs:* The number of bugs for which at least one generated MTC fails on the buggy version and passes on the fixed version.

4.4.2 Experiment Result. As shown in Table 2, MR-Coupler (*DeepSeek-V3.1-Think*) performed the best, successfully revealing 22 real-world bugs with a bug-revealing MTC percentage of 6.53%. Specifically, it revealed 15 more bugs and achieved a 98.21% improvement in bug-revealing MTC percentage compared with the baseline. When combined with other models, MR-Coupler consistently outperforms baselines, revealing 9∼15 additional bugs and achieving an average 67.65% increase in bug-revealing MTC rate.

Based on manual inspection, we attribute the improvement to two main factors: (i) *Relevance-aware MR construction.* Some bugs can only be triggered by the MRs that couple specific method pairs, and MR-Coupler can generate such MRs through its coupling analysis step. For example, a bug in the "producer" method `randomRepo` can be revealed when coupled with the "consumer" method `repos`, which accesses the coordinates of the newly created repository [20]. Both methods invoke the same

API `MkRepo` and access the same fields (`storage` and `self`), allowing MR-Coupler to identify their relevance and generate an effective MR. (ii) *Input augmentation.* Some bugs require specific input to trigger. By applying additional inputs (Section 3.3.1), MR-Coupler exercises a broader range of program behaviors and uncovers such corner-case bugs. For example, a bug in `previousClearBit` manifests only when processing inputs near array boundaries (e.g., `int i=1«16`) [44]. In addition, the performance comparison among the evaluated LLMs shows that *DeepSeek-V3.1-Think* revealed the highest number of bugs, likely due to its superior reasoning ability. By analyzing the code of target methods and understanding the relevance between methods, it can reason about potential fault-prone scenarios and construct MRs that expose them.

By analyzing the overlap of bugs revealed by MR-Coupler with different LLMs, we observed that a total of 28 unique bugs were revealed. 8 of the 28 bugs were detected by all models, and both *DeepSeek-V3.1-Think* and *Qwen3-coder-Flash* uniquely revealed two additional bugs. This suggests that *combining models could further improve the bug-revealing capability.* This is an interesting strategy for future enhancement.

We also observed that 19 out of 22 bugs revealed by *DeepSeek-V3.1-Think* were detected by more than one distinct MTC. For example, a bug in a "`multiply`" method could be revealed by multiple MRs, such as $a = divide(multiply(a, b), b)$ or $multiply(subtract(a, b), c) = subtract(multiply(a, c), multiply(b, c))$ [31]. This highlights that *MR diversity plays a role in enhancing the bug-revealing capability.*

Failure analysis. While MR-Coupler successfully detected 22 (44%) real bugs originally revealed by human-written MTCs, it failed to expose some others. A major reason is that certain methods under test are highly domain-specific and implement complex business logic. In such cases, simply providing the code of the method is insufficient for an LLM to fully understand its functionality and intended specification. For example, in a bug related to "compaction file metrics" in Apache IoTDB [18], understanding the expected behavior of the `doCompaction` method requires deeper module-level or even project-level knowledge. Automatically extracting such background context and enabling an LLM to reason about domain-specific business logic remains a challenging and promising direction for future research. In other cases, constructing inputs required access to external resources, such as environment variables or specific file contents [59]. When no concrete examples were retrieved in the project under test, MR-Coupler generated MTCs failed to set up that, resulting in non-executable or non-bug-revealing tests. Automatically and completely retrieving and adapting such project-level context for test generation is an open challenge for future work.

Answer to RQ2: MR-Coupler can successfully detect 22 (out of 50) real-world bugs originally discovered by human-written MTCs. However, some unrevealed bugs are rooted in domain-specific business logic, requiring model-level or even project-level context to construct bug-revealing MTCs. Effectively leveraging such context remains an open challenge for future work.

4.5 RQ3: Ablation Study on MR-Coupler

4.5.1 Experiment Setup. This RQ aims to evaluate the contribution of major steps in MR-Coupler to its overall effectiveness in generating valid and bug-revealing MTCs. We use the same tasks and metrics as in RQ1 (validity) and RQ2 (bug-revealing capability) We created three ablated variants of MR-Coupler (v_1, v_2, and v_3) by ablating three steps to analyze their contribution. We chose MR-Coupler built with *DeepSeek-V3.1-Think* which achieves the best result in RQ1 and RQ2 (Sections 4.3 and 4.4). The variants are as follows:

Table 3. Ablation Study on MR-Coupler (*DeepSeek-V3.1-Think*)

Metric	MR-Coupler	v_1: w/o func. context	v_2: w/o MTC expansion	v_3: w/o MTC validation
Num. of Generated TC	4151	3367	4324	4146
Pct. of Executable MTC	88.44%	63.86% (-27.80%)	88.34% (-0.10%)	91.65% (3.94%)
Pct. of Valid MTC	83.57%	**56.28% (-32.65%)**	81.38% (-2.62%)	83.04% (-0.63%)
Num. of Successful Task	98	**86 (-12.24%)**	98 (0.00%)	97 (-1.02%)
Pct. of False Alarm	5.50%	**11.86 (115.53%)**	5.73% (4.13%)	**9.39% (70.65%)**
Pct. of Bug-revealing MTC	7.77%	**3.37% (-56.62%)**	**3.89% (-49.92%)**	14.06% (81.04%)
Num. of Revealed Bugs	22	**12 (-45.45%)**	**13 (-40.91%)**	24 (9.09%)

- v_1**: MR-Coupler w/o Coupling Analysis.** This variant disables the *Coupling Analysis* step (Section 3.1), meaning no functionally coupled methods and corresponding invocation examples are provided as context to LLMs during the MTC generation.
- v_2**: MR-Coupler w/o MTC Amplification.** This variant disables the *Input Agumentation* step (Section 3.3.1), thus no additional inputs are generated to amplify MTCs.
- v_3**: MR-Coupler w/o MTC Validation.** This variant disables the *Validation and Filtering* step (Section 3.3.2), meaning all generated MTCs are retained without filtering.

4.5.2 Experiment Result. As shown in Table 3, disabling *Coupling Analysis* (v_1) led to a 32.65% decrease in valid MTC rate and a 56.62% decrease in bug-revealing rate. This suggests that leveraging functional coupling as an explicit hint is crucial for generating valid MTCs and reducing hallucinations. This aligns with the findings in RQs of validation and bug-revealing capability.

When disabling *Input Agumentation* (v_2), the revealed bugs significantly decreased by 40.91% (from 22 to 13). This highlights that applying generated MRs to additional inputs strengthens generated MTCs by exercising a wider range of program behaviors. Nevertheless, even without input augmentation, MR-Coupler revealed more bugs compared with the baseline, indicating that MRs over multiple methods already contribute to bug revealing, and MTC amplification further boosts the bug-revealing rate by 89.94% (from 3.89% to 7.77%).

Disabling *Validation and Filtering* (v_3) increased the false-alarm percentage by 70.65% (from 5.5% to 9.39%), indicating the effectiveness of the validation step in mitigating the false alarm issue. The slight increase in bug-revealing rate is because some bug-revealing MTCs are filtered out together with invalid ones. In some cases, both invalid and bug-revealing MTCs fail on both the original and mutated versions (0% pass rate), or valid MTCs fail to kill any mutants (100% pass rate on both), making mutation analysis based validation unable to distinguish them. A possible mitigation is to generate more diverse inputs to improve the mutant-killing capability.

Answer to RQ3: Each of the three steps uniquely enhances MR-Coupler's effectiveness. Functional coupling helps LLMs to generate more valid MTCs, MTC amplification augments the input to reveal more bugs, and mutation analysis based validation filters nearly half of the false alarms (reducing the rate from 9.39% to 5.5%).

4.6 RQ4: Similarity to human-written MTCs

4.6.1 Experiment Setup. Human-written MTCs represent well-established practices for constructing MRs, including the selection of method pairs as well as the input and output relation construction. This RQ evaluates whether the MTCs generated by MR-Coupler can mimic these practices by checking if they encode the same *MR-skeletons* as human-written ones. This can demonstrate the potential of MR-Coupler to assist developers in MTCs construction, facilitating developers in integrating the generated tests into their codebase and easing subsequent maintenance. We use the same 100 evaluation tasks as in RQ1.

Table 4. Similarity of MR-Coupler-generated MTCs to human-written MTCs

Metric	GPT-4o-mini		Qwen3-coder-Flash		Deepseek-V3.1		Deepseek-V3.1-Think		**Improv.**
	Baseline	MR-Coupler	Baseline	MR-Coupler	Baseline	MR-Coupler	Baseline	MR-Coupler	
L1: Method-Pair consistency	61%	89% (+45.90%)	61%	86% (+40.98%)	74%	87% (+17.56%)	81%	92% (+13.58%)	**29.51%**
L2: MR-Skeleton consistency	46%	85% (+84.78%)	46%	84% (+82.61%)	55%	84% (+52.73%)	65%	90% (+38.46%)	**64.65%**

According to the definition of MTC in Section 2, an MR-skeleton consists of three core components: input relation, execution, and output relation. (i) *Input Relation:* the input transformation (e.g., API calls) applied to generate follow-up inputs, if applicable. (ii) *Execution:* the MR involved method pair (e.g., `<encryptText, decryptText>` in Listing 1). (iii) *Output Relation:* the assertion type (e.g., `assertEquals`) and the involved elements (e.g., source input, source output, follow-up output) to verify the output relation. For example, `assertEquals(plainText, decryptedText)` uses the assertion type `assertEquals`, with the involved elements source input (`plainText`) and follow-up output (`decryptedText`). Considering that the same output relation can be implemented in multiple ways, we normalize assertions for ease of comparison. Specifically, we normalize assertions to comparable assertions [48]. For example, boolean-style `assertTrue(x.equals(y))` and `assertFalse(x.equals(y))` are normalized to `assertEquals(x, y)` and `assertNotEquals(x, y)`, respectively. More details can be found in MR-Coupler's artifact [42].

Based on the definition of MR-skeleton, we take the human-written MTCs as the ground truth, and measure the similarity at two levels:

- L1: Method-Pair consistency: the proportion of target methods where at least one generated MTC couples the *same method pair* as the human-written MTC.
- L2: MR-Skeleton consistency: the proportion of target methods where at least one generated MTC encodes the same MR-skeleton as the human-written MTC, i.e., matching the input transformation, method pair, and output relation assertion type and elements. The MTCs satisfying L2 must satisfy L1 as well.

4.6.2 Experiment Result. Table 4 shows that MR-Coupler-generated MTCs can match the human-coupled method pairs for 86~92 target methods and encode the same MR-skeletons for 84~90. Compared to the baseline, MR-Coupler improves method-pair consistency by 29.51% and full MR-skeleton consistency by 64.65%. These results highlight the effectiveness of MR-Coupler's coupling analysis based on patterns of relevant intention, similar implementation behavior, and potential state interference, MR-Coupler identified most functionally coupled methods used in human-written MTCs. The high MR-skeleton consistency further demonstrates MR-Coupler's potential to assist developers in MTCs construction, integration, and maintenance.

Failure analysis. Despite the overall high consistency, MR-Coupler (*DeepSeek-V3.1-Think*) missed eight tasks in identifying the same method pairs and failed to encode the same MR-skeleton in ten tasks. Our inspection revealed three main causes: (i) some developer-selected method pairs exhibit implicit relevance in the code, such as `a2q` paired with `readAndWrite` for JSON serialization [17]; (ii) MR-Coupler-generated MTCs sometimes construct correct but different MRs, e.g., `cosineSimilarity` asserting equality for identical vectors versus inequality for distinct vectors [12]; and (iii) some inconsistencies arise from equivalent but differently expressed assertions, such as `assertEquals(x,y)` versus `assertTrue(a.equals(x)&&a.equals(y))` [26].

Table 5. Performance of MR-Coupler (GPT-4o-mini) on 100 Target Methods Before or After the Cut-Off Date

Target methods	Validity			Similarity	
	Num. of Successful Tasks	Pct. of Valid MTCs	Pct. of False alarm	L1: Method-Pair consistency	L2: MR-Skeleton consistency
Before Cut-off	92	71.84%	14.16%	89	85
After Cut-off	94	73.17%	9.52%	85	79

Answer to RQ4: The coupling analysis in MR-Coupler can identify most of the relevant methods used in human-written MTCs. MR-Coupler can achieve over 90% MR-skeleton consistency with human-written MTCs. This demonstrates MR-Coupler's potential to assist developers in MTC construction, facilitating developers in integrating and maintaining generated MTCs into their codebase.

4.7 Threats to Validity

Representativeness of LLMs. Since MR-Coupler relies on LLMs for MTC and input generation, one potential threat is whether our findings based on the selected LLMs are representative. To mitigate this threat, according to the EvalPlus leaderboard [16], we include representative LLMs from three well-known LLM families, i.e., *GPT-4o mini* from OpenAI [30],*Qwen3-coder-Flash* from Alibaba [1], *DeepSeek-V3.1* and *DeepSeek-V3.1-Think* from DeepSeek [11].

Data Contamination. A potential threat to our study is the data contamination issue, where some of the target programs or MTCs in our evaluation dataset may have been included in the pretraining data of the evaluated LLMs. If such memorization occurs, the models could gain an unfair advantage, thereby biasing the evaluation results. To mitigate this threat, we followed the same collection procedure described in Section 4.1 to construct a *after-cutoff* dataset of entries after the training cutoff date of GPT-4o-mini (October 2023 [21]). As shown in Table 5, MR-Coupler achieved slightly lower similarity to human-written MTCs but a higher percentage of valid MTC compared to the pre-cutoff dataset. This indicates that MR-Coupler's effectiveness still holds for subjects after the cut-off date, and not simply an artifact of training-data memorization.

Representativeness of Subjects. A potential threat is whether our findings generalize to subjects of different projects. To mitigate this, we adopted the strategy from existing studies [43, 47, 48] to include representative Java projects and evaluated MR-Coupler based on these projects (Section 4.1).

Quality of Ground Truths. We use human-written MTCs as ground truth, and take the fixed or latest versions of target programs as bug-free for answering RQs of validity, bug-revealing, and similarity. There is a potential threat regarding the quality of these ground truths. To mitigate this threat, we applied three filtering criteria to select the valid and high-quality MTCs and corresponding target methods (Section 4.1).

5 RELATED WORK

Constructing effective MRs is a critical step in conducting MT, and numerous approaches have been proposed to facilitate this process.

LLM-Based MR Generation. Recently, several studies have explored using LLMs to generate MRs. Xu et al. [47] proposed an LLM-based test to automatically deduce the input relation from the pairs of hard-coded source and follow-up inputs. Shin et al. [35] employed LLMs to derive MRs from requirement specifications and translate them into an MR-specific language. Their approach works in two phases: (i) deriving metamorphic relations from a requirements document, and then (ii) converting MRs into SMRL (a domain-specific language for MRs). Zhang et al. [55] developed a human-AI hybrid MT framework that uses LLMs and predefined MR patterns to generate MRs for autonomous driving systems (ADSs). The two approaches are semi-automated and rely on human experts to select and refine generated MRs.

Some studies evaluated the effectiveness of LLMs in generating MR. Zhang et al. [56] conducted a pilot study using ChatGPT (3.5) for MR generation in ADS testing, and provided a methodology for generating MRs. Zhang et al. [54] evaluated LLMs (GPT-3.5 and GPT-4) on MR discovery across 37 subjects, finding that 4.6 38.6% of new MRs were rediscovered but only 29.9 43.8% of generated MRs were valid. These studies show the effectiveness of LLMs in generating MRs, but also challenges, such as a high invalidity rate of generated MRs.

Our approach (MR-Coupler) makes use of LLM's reasoning capability to come up with MRs based on functionally coupled methods and then generate concrete MTCs. MR-Coupler further mitigates the invalidity issue by validating generated MTCs via mutation analysis, providing an end-to-end automated and self-validating solution.

Traditional MR Generation Approaches. Prior to LLMs, most MR generation techniques relied on search-based, pattern-based, genetic-programming-based, or heuristic approaches. Ayerdi et al. [2, 4] and Terragni et al. [40] proposed approaches to generate MRs via genetic programming, but assuming the regression testing scenario. Zhang et al. [52] and Zhang et al. [53] proposed search-based approaches to inferring MRs for numeric programs. Zhou et al. [58] and Segura et al. [34]'s approaches identify MRs based on a set of predefined patterns. These approaches generated MRs are specific to domains or certain pre-defined patterns. Nolasco et al. [29] proposed MemoRIA to identify equivalence MRs from the documentation. Recently, Xu et al. [48] leveraged a new source (i.e., existing test cases) to automatically derive MRs. However, their approaches rely on rare resources (i.e., documents or tests embedded with MRs).

Compared with these approaches, our approach MR-Coupler is a fully automatic and domain-agnostic technique that generates MTCs directly from the target program. Not having the limitations of these approaches, MR-Coupler does not rely on resources like MR-embedded documents or tests, and is not restricted to the regression testing scenario.

6 CONCLUSION

This paper presents MR-Coupler, a fully automated approach to generate MTCs directly from the target program via functional coupling analysis. MR-Coupler first identifies functionally coupled method pairs based on relevant intentions, similar implementation behavior, and potential state interference. It then employs LLMs to generate concrete MTCs and refines them based on execution feedback. Finally, MR-Coupler amplifies and then validates the MTCs based on mutation analysis.

Our evaluation shows that MR-Coupler effectively generates valid MTCs for 98% of tasks and successfully reveals 22 confirmed bugs, improving valid MTC generation by 64.90%, and reducing false alarms by 36.56% compared to baselines. Moreover, MR-Coupler-generated achieves high consistency with human-written MR skeletons, demonstrating its potential to assist or even partially replace developers in constructing effective MTCs across diverse domains.

In summary, our work offers valuable insights into constructing metamorphic relations without relying on rare resources, such as human experts or high-quality specifications. It provides both a practical tool and a useful dataset for future research endeavors.

7 DATA AVAILABILITY

MR-Coupler and our experimental data are available at our website [42].

REFERENCES

[1] Alibaba. 2025. *Qwen3-coder.* Retrieved September 1, 2025 from https://qwenlm.github.io/blog/qwen3-coder/

[2] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. 2021. Generating metamorphic relations for cyber-physical systems with genetic programming: an industrial case study. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1264–1274.

[3] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. 2022. Evolutionary generation of metamorphic relations for cyber-physical systems. In *GECCO '22: Genetic and Evolutionary Computation Conference, Companion Volume, Boston, Massachusetts, USA, July 9 - 13, 2022*, Jonathan E. Fieldsend and Markus Wagner (Eds.). ACM, 15–16. https://doi.org/10.1145/3520304.3534077

[4] Jon Ayerdi, Valerio Terragni, Gunel Jahangirova, Aitor Arrieta, and Paolo Tonella. 2024. GenMorph: Automatically Generating Metamorphic Relations via Genetic Programming. *IEEE Transactions on Software Engineering* (2024), 1–12.

[5] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *J. Syst. Softw.* 181 (2021), 111041. https://doi.org/10.1016/J.JSS.2021.111041

[6] Jialun Cao, Meiziniu Li, Yeting Li, Ming Wen, Shing-Chi Cheung, and Haiming Chen. 2022. SemMT: A Semantic-Based Testing Approach for Machine Translation Systems. *ACM Transactions on Software Engineering and Methodology* 31, 2 (2022), 34e:1–34e:36.

[7] Jialun Cao, Wuqi Zhang, and Shing-Chi Cheung. 2024. Concerned with Data Contamination? Assessing Countermeasures in Code Language Model. *CoRR* abs/2403.16898 (2024). arXiv:2403.16898

[8] Songqiang Chen, Shuo Jin, and Xiaoyuan Xie. 2021. Testing Your Question Answering Software via Asking Recursively. In *International Conference on Automated Software Engineering*. IEEE, 104–116.

[9] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1 (2018), 4:1–4:27. https://doi.org/10.1145/3143561

[10] Tsong Yueh Chen, Pak-Lok Poon, and Xiaoyuan Xie. 2016. METRIC: METamorphic Relation Identification based on the Category-choice framework. *J. Syst. Softw.* 116 (2016), 177–190. https://doi.org/10.1016/j.jss.2015.07.037

[11] DeepSeek. 2025. *DeepSeek-V3.1.* Retrieved September 1, 2025 from https://api-docs.deepseek.com/news/news250821

[12] Diennea. 2025. *SimplerPlannerTest.* https://github.com/diennea/herddb/blob/master/herddb-core/src/test/java/herddb/sql/SimplerPlannerTest.java

[13] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *nternational Conference on Software Engineering*. ACM, 81:1–81:13.

[14] Eclipse EE4J. 2025. *CompactFormatterTest.java.* https://github.com/eclipse-ee4j/angus-mail/blob/master/providers/angus-mail/src/test/java/org/eclipse/angus/mail/util/logging/CompactFormatterTest.java

[15] Aryaz Eghbali and Michael Pradel. 2024. De-Hallucinator: Iterative Grounding for LLM-Based Code Completion. *CoRR* abs/2401.01701 (2024). arXiv:2401.01701

[16] Evalplus. 2025. *leaderboard.* Retrieved September 1, 2025 from https://evalplus.github.io/leaderboard.html

[17] FasterXML. 2025. *JUnit5TestBase.* https://github.com/FasterXML/jackson-core/blob/2.x/src/test/java/com/fasterxml/jackson/core/JUnit5TestBase.java

[18] Apache IoTDB. 2025. *IoTDB Issue #13691.* https://github.com/apache/iotdb/pull/13691

[19] JavaParser. 2024. *JavaParser.* Retrieved June 6, 2024 from https://javaparser.org/

[20] Jcabi. 2025. *GitHub Commit 777a078913.* https://github.com/jcabi/jcabi-github/commit/777a078913

[21] Knowledge Cutoff Information of GPT-4o-mini [n. d.]. https://community.openai.com/t/introducing-gpt-4o-mini-in-the-api/871594

[22] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Conference on Programming Language Design and Implementation*. ACM, 216–226.

[23] Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. 2023. Chain of Code: Reasoning with a Language Model-Augmented Code Emulator. *CoRR* abs/2312.04474 (2023). arXiv:2312.04474

[24] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, and et al. 2022. Competition-Level Code Generation with AlphaCode. *CoRR* abs/2203.07814 (2022). arXiv:2203.07814

[25] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. 2014. How Effectively Does Metamorphic Testing Alleviate the Oracle Problem? *IEEE Transactions on Software Engineering* 40, 1 (2014), 4–22.

[26] LocationTech. 2025. *NTV2Test.* https://github.com/locationtech/proj4j/blob/master/core/src/test/java/org/locationtech/proj4j/datum/NTV2Test.java

[27] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2023. Fuzzing Deep Learning Compilers with HirGen. In *International Symposium on Software Testing and Analysis*. ACM, 248–260.

[28] Major. 2025. *Mutation Testing*. https://mutation-testing.org/
[29] Agustín Nolasco, Facundo Molina, Renzo Degiovanni, Alessandra Gorla, Diego Garbervetsky, Mike Papadakis, Sebastián Uchitel, Nazareno Aguirre, and Marcelo F. Frias. 2024. Abstraction-Aware Inference of Metamorphic Relations. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 450–472.
[30] OpenAI. 2025. *GPT-4o mini*. Retrieved September 1, 2025 from https://platform.openai.com/docs/models/gpt-4o-mini
[31] Optimatika. 2025. *OjAlgo Issue #49*. https://github.com/optimatika/ojAlgo/issues/49
[32] Apache Sedona. 2025. *QuadTreeTest*. https://github.com/apache/sedona/blob/master/spark/common/src/test/java/org/apache/sedona/core/spatialPartitioning/quadtree/QuadTreeTest.java
[33] Sergio Segura, Gordon Fraser, Ana Belén Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Trans. Software Eng.* 42, 9 (2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875
[34] Sergio Segura, José Antonio Parejo, Javier Troya, and Antonio Ruiz Cortés. 2018. Metamorphic Testing of RESTful Web APIs. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1083–1099.
[35] Seung Yeob Shin, Fabrizio Pastore, Domenico Bianculli, and Alexandra Baicoianu. 2024. Towards Generating Executable Metamorphic Relations Using Large Language Models. In *Quality of Information and Communications Technology - 17th International Conference on the Quality of Information and Communications Technology, QUATIC 2024, Pisa, Italy, September 11-13, 2024, Proceedings (Communications in Computer and Information Science, Vol. 2178)*, Antonia Bertolino, João Pascoal Faria, Patricia Lago, and Laura Semini (Eds.). Springer, 126–141. https://doi.org/10.1007/978-3-031-70245-7_9
[36] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications,*. ACM, 849–863.
[37] Chang-Ai Sun, Yiqiang Liu, Zuoyi Wang, and W. K. Chan. 2016. μMT: a data mutation directed metamorphic relation acquisition methodology. In *International Workshop on Metamorphic Testing*. ACM, 12–18.
[38] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2024. ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. *IEEE Transactions on Software Engineering* (2024), 1–19.
[39] Gradoop Team. 2025. *CNFEstimation*. https://github.com/dbs-leipzig/gradoop/blob/develop/gradoop-temporal/src/main/java/org/gradoop/temporal/model/impl/operators/matching/single/cypher/planning/estimation/CNFEstimation.java
[40] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary Improvement of Assertion Oracles. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1178–1189.
[41] TheAlgorithms. 2025. *AESEncryptionTest*. https://github.com/TheAlgorithms/Java/blob/master/src/test/java/com/thealgorithms/ciphers/AESEncryptionTest.java
[42] MR-Coupler. 2025. *MR-Coupler*. Retrieved September 2, 2025 from https://mr-coupler.github.io/
[43] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects. In *International Conference on Software Maintenance and Evolution*. IEEE, 35–45.
[44] Brett Wooldridge. 2025. *SparseBitSet Issue #13*. https://github.com/brettwooldridge/SparseBitSet/issues/13
[45] Xiaoyuan Xie, Shuo Jin, and Songqiang Chen. 2023. qaAskeR$^+$: a novel testing method for question answering software via asking recursive questions. *Automated Software Engineering* 30, 1 (2023), 14.
[46] Xiaoyuan Xie, Shuo Jin, Songqiang Chen, and Shing-Chi Cheung. 2024. Word Closure-Based Metamorphic Testing for Machine Translation. *ACM Transactions on Software Engineering and Methodology* (jul 2024).
[47] Congying Xu, Songqiang Chen, Jiarong Wu, Shing-Chi Cheung, Valerio Terragni, Hengcheng Zhu, and Jialun Cao. 2024. MR-Adopt: Automatic Deduction of Input Transformation Function for Metamorphic Testing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 557–569. https://doi.org/10.1145/3691620.3696020
[48] Congying Xu, Valerio Terragni, Hengcheng Zhu, Jiarong Wu, and Shing-Chi Cheung. 2024. MR-Scout: Automated Synthesis of Metamorphic Relations from Existing Test Cases. *ACM Transactions on Software Engineering and Methodology* 33, 6 (2024), 150.
[49] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. *CoRR* abs/2404.14646 (2024). arXiv:2404.14646
[50] Yuanyuan Yuan, Shuai Wang, Mingyue Jiang, and Tsong Yueh Chen. 2021. Perception Matters: Detecting Perception Failures of VQA Models Using Metamorphic Testing. In *Conference on Computer Vision and Pattern Recognition*. Computer Vision Foundation / IEEE, 16908–16917.
[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] Bo Zhang, Hongyu Zhang, Junjie Chen, Dan Hao, and Pablo Moscato. 2019. Automatic Discovery and Cleansing of Numerical Metamorphic Relations. In *IEEE International Conference on Software Maintenance and Evolution*. IEEE, 235–245.
[53] Jie Zhang, Junjie Chen, Dan Hao, Yingfei Xiong, Bing Xie, Lu Zhang, and Hong Mei. 2014. Search-based inference of polynomial metamorphic relations. In *ACM/IEEE International Conference on Automated Software Engineering*. ACM, 701–712.
[54] Jiaming Zhang, Chang-Ai Sun, Huai Liu, and Sijin Dong. 2025. Can Large Language Models Discover Metamorphic Relations? A Large-Scale Empirical Study. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2025, Montreal, QC, Canada, March 4-7, 2025*. IEEE, 24–35. https://doi.org/10.1109/SANER64311.2025.00011
[55] Yifan Zhang, Tsong Yueh Chen, Matthew Pike, Dave Towey, Zhihao Ying, and Zhi Quan Zhou. 2025. Enhancing autonomous driving simulations: A hybrid metamorphic testing framework with metamorphic relations generated by GPT. *Inf. Softw. Technol.* 187 (2025), 107828. https://doi.org/10.1016/J.INFSOF.2025.107828
[56] Yifan Zhang, Dave Towey, and Matthew Pike. 2023. Automated Metamorphic-Relation Generation with ChatGPT: An Experience Report. In *47th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2023, Torino, Italy, June 26-30, 2023*. IEEE, 1780–1785. https://doi.org/10.1109/COMPSAC57700.2023.00275
[57] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. *Proc. ACM Softw. Eng.* 2, ISSTA (2025), 481–503. https://doi.org/10.1145/3728894
[58] Zhi Quan Zhou, Liqun Sun, Tsong Yueh Chen, and Dave Towey. 2020. Metamorphic Relations for Enhancing System Understanding and Use. *IEEE Transactions on Software Engineering* 46, 10 (2020), 1120–1154.
[59] Zingg. 2025. *Zingg Issue #60*. https://github.com/zinggAI/zingg/issues/60