Language Models as Architectural Gatekeepers: Automating Conformance Checking from Natural Language

Andrielly Lucena
Department of Systems and Computing, Federal University of Campina Grande
Campina Grande, Brazil
andriellylucena@copin.ufcg.edu.br

Everton L. G. Alves
Department of Systems and Computing, Federal University of Campina Grande
Campina Grande, Brazil
everton@computacao.ufcg.edu.br

João Brunet
Department of Systems and Computing, Federal University of Campina Grande
Campina Grande, Brazil
joao.arthur@computacao.ufcg.edu.br

ABSTRACT

Ensuring that implemented code adheres to its intended architecture (architectural conformance) remains a critical challenge in software engineering. While formal verification tools exist, their use is hindered by the overhead of explicitly defining formal architectural specifications. At the same time, valuable architectural decisions and design constraints are often embedded in informal channels, such as pull request discussions and issue trackers, where they are expressed in natural language rather than formal specifications. In this paper, we explore the potential of Large Language Models (LLMs) to bridge this gap by investigating whether they can transform informal design rules, based on real development discussions on GitHub, into design tests — executable conformance checkers. Rather than relying on formal models, our approach focuses on deriving design tests from implicit architectural decisions discussed in natural language. To investigate this, we conducted a preliminary empirical study to generate 30 design tests representing rules for 6 design patterns. Our results show the potential of such a strategy, as 96.67% of the generated design tests execute successfully. Furthermore, 63.33% correctly assert expected behaviors, and 76.67% accurately reflect the intended architectural rules. This approach has the potential to simplify the conformance checking process by reducing the need to manually write formal specifications and tests. By leveraging existing development discussions, it makes the process more accessible and less time-consuming, and supports early identification of architectural violations during development.

KEYWORDS

Large Language Models, Design Rules, Architectural Conformance, Design Tests

1 Introduction

Software architecture is a fundamental aspect of software development. It serves as an abstraction that communicates to stakeholders how the system is structured, reflecting the earliest and most critical design decisions. [3] This architectural blueprint typically imposes constraints on how code entities should be organized within the system. However, as the system evolves, these constraints are often no longer followed, leading to a gap between the planned design and the actual implementation. [4] This phenomenon, commonly referred to as architecture drift or erosion, can lead to several issues and is widely discussed in the field of software architecture [21][2].

Due to its significance, various methods and tools have been developed to detect architecture drift and erosion in software systems [19][7][8][10][18]. One prior study [5] proposed an approach that uses JUnit tests to verify conformance to design constraints. The authors developed Design Wizard, a Java library that provides an interface for querying and manipulating a system's structural design. The idea of design tests, as introduced by the authors, is compelling because it leverages tools that developers already use regularly, minimizing the need for additional learning or context.

However, these approaches still require the intended architecture and its rules to be defined in specific formats or languages in order to be verifiable against the code. This documentation and testing effort may impose an additional burden on developers and, in turn, contribute to the architectural drift and erosion.

Nevertheless, design decisions are often discussed — even if not formally documented. A previous study [6] found that developers frequently address architectural concerns through communication channels such as issues, pull request comments, and commit messages.

Based on this, our work explores the feasibility of automatically generating design tests from informal development discussions, using Large Language Models (LLMs). By leveraging development discussions expressed in natural language, our approach avoids the need to manually define architectural models or formal rules. To investigate this, we simulate sentences extracted from real-world discussions and evaluate the ability of LLMs to interpret natural language and translate it into automated design tests using the Design Wizard library [5].

2 Background

2.1 Design Tests and Design Wizard

The concept of design tests was presented in a previous study [5] as a means of verifying specific design properties within a system. The authors proposed generating the tests in Java using JUnit and developed the Design Wizard library to facilitate this process. This library provides methods for inspecting and verifying properties and relationships among system entities such as classes, methods, packages, and attributes.

Listing 1 shows an example of a design test using Design Wizard. The code checks whether the class `ConcreteStrategyA` extends the class `Strategy`. These validations are performed using methods from the library, which require the path to the directory containing the system's compiled class bytecode. With this information, the tool can check the structure and relationships of the system.

Listing 1: Design test example

```java
@Test
public void testConcreteImplementsStrategy() {
  DesignWizard dw = new DesignWizard("bin");
  ClassNode classNode =
    dw.getClass("com.example.ConcreateStrategyA");
  ClassNode classNode2 =
    dw.getClass("com.example.Strategy");
  assertTrue(classNode.extendsClass(classNode2));
}
```

A suite composed of Design Wizard tests is the expected output in our pipeline, since they translate architectural constraints into executable code.

In this work, we adopt a structural view of software architecture, focusing on recurring constraints that govern the modular organization of a system (such as the use of design patterns, inheritance, and allowed dependencies between classes and packages) in a way that is compatible with widely accepted definitions of software architecture [3]. While the generated tests resemble unit tests in format, they are intended to check these structural rules rather than detailed implementation logic. As shown in Listing 1, they validate properties that often emerge from informally discussed architectural guidelines during development.

2.2 Large Language Models

Large Language Models (LLMs) are deep learning models trained on vast amounts of text data and capable of handling natural language tasks such as text generation, summarization, and translation. [17] Most LLMs are based on the Transformer architecture [20], which revolutionized machine learning by enabling more efficient and accurate model training.

In addition to achieving excellent results in natural language tasks, LLMs have been fine-tuned for more specialized domains, including code-related tasks [9]. This has enabled new opportunities to automate or support various software development processes, such as documentation, testing, and code generation.

In our work, we explore the ability of LLMs to bridge natural language and code by generating design tests from natural language design discussions.

2.3 Retrieval Augmented Generation

Retrieval Augmented Generation (RAG) [12] is a technique that enhances the performance of LLMs by providing them with external contextual information that may not have been deeply covered during training. This approach involves constructing an external knowledge base on the topic of interest and using a retrieval mechanism to extract the most relevant content in response to a query. The retrieved information is then supplied to the LLM, enabling more accurate and context-aware outputs.

In this work, we apply the RAG technique to supply the model with contextual knowledge about the Design Wizard library. This allowed the LLM to correctly use the library when generating design tests.

3 Study Design

3.1 Research Questions

Our main goal is to assist developers in verifying architectural constraints by automatically generating design tests. These tests aim to check compliance with rules related to the system's architecture. Therefore, we executed an empirical study to explore the feasibility of using LLMs to generate design tests based on natural language design rules. To guide our investigation, we formulated the following research questions:

RQ$_1$: Can LLMs effectively generate design tests? This question evaluates whether LLMs can generate JUnit tests using Design Wizard to verify specific architectural rules. To answer this, we analyze the generated test code for compilation and assertion errors.

RQ$_2$: How well do these generated tests adhere to the corresponding design rules in terms of fidelity and correctness? This question examines whether the tests adequately verify the constraints described by the rules and whether they are complete within the defined rule scope.

RQ$_3$: What is the effort to fix the design tests that have problems? This question aims to verify how much of the generated tests with error needs to be revised and/or can help conformance checking.

3.2 Methodology

Figure 1 presents an overview of our empirical study. Each phase is described in more detail below.

3.2.1 Rules Generation The first step involved gathering examples of design rules. As the subject of the study, we selected a repository of Design Pattern implementations [1], which contains 24 Java mini-projects, each implementing a different design pattern to demonstrate how these patterns can be applied in code.

In a realistic scenario, design rules would typically be extracted from commits or code review messages. However, since this study focuses on exploring the capabilities of LLMs specifically for generating design tests, we generated synthetic rules using an LLM to simulate how developers commonly express architectural constraints in practice. We ensured that these rules reflect realistic architectural scenarios and describe meaningful structural or behavioral constraints. Listing 2 shows an example of a rule extracted from an actual development discussion, taken from the dataset produced by Brunet et al. [6], which served as inspiration for our rule generation. The Design Pattern repository was chosen because it allowed us to generate rules without needing to provide additional context about business logic or system structure, as the patterns themselves are widely known and well-documented.

Listing 2: Example rule extracted from real discussion

```
I think its important to specify that the
    developer must implement ServiceIface and not
    Iface to make the example clearer
```

Using the OpenAI GPT-4o model [15], we generated 30 design rule sentences covering six different patterns. The prompt instructed the model to produce sentences based on examples of real-world design-related discussions from previous research [6].

[1]https://github.com/clarck/DesignPatterns

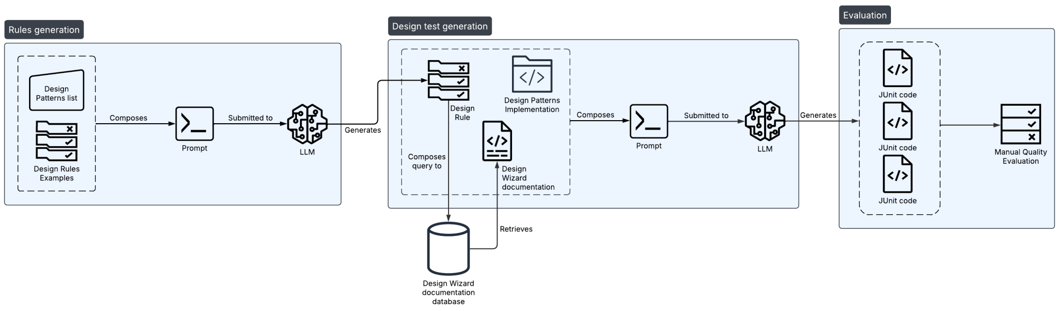

Figure 1: Overview of the methodological approach

We selected GPT-4o due to its state-of-the-art performance in natural language understanding and generation tasks, as well as its ability to generate coherent, semantically rich, and contextually relevant text [11]. These characteristics make it well-suited for synthesizing design rules that mimic how developers naturally express architectural concerns. Furthermore, GPT-4o is widely accessible and well-documented, which contributes to the reproducibility of our study. The script and prompt used for this generation process are available in our GitHub repository [**?**]. An example of a generated rule for the Strategy Pattern [2] is shown in Listing 3.

Listing 3: Example of design rule for the Strategy Pattern.

```
Each concrete implementation must encapsulate a
    single algorithm and be completely independent
     of the other strategies.
```

To better analyze the quality and correctness of the generated tests, we classified the design rules into three difficulty levels. This categorization aims to provide insight into whether the complexity of a rule affects the model's ability to generate effective tests.

The difficulty levels were defined based on three main aspects: i) the number of entities (classes, methods, fields) involved in the rule; ii) the clarity regarding which entities are being referenced; and iii) the clarity regarding what needs to be verified among these entities. Table 1 shows the distribution of the rules across the three levels.

Level	Number of Rules
1	14
2	12
3	4

Table 1: Distribution of rules by difficulty level

3.2.2 *Design Test Generation*

After generating the set of design rules, we configured the techniques to be used in the experiments. For this study, we chose the Design Wizard [5] and JUnit for architectural conformance checking. Since the Design Wizard is a lesser-known library, existing language models were not trained with enough data to recognize it. Therefore, we employed the Retrieval-Augmented Generation (RAG) technique to create a knowledge source about the Design Wizard. We indexed its documentation, available in the GitHub repository [3], into a database. The documentation includes information about available methods and classes for checking design aspects of code.

Since the documentation was in HTML format, we performed preprocessing to extract the main text of the page, removing elements that did not contribute to the content, such as footer and sidebar texts. After this step, we stored the texts in a PostgreSQL database [16], using a vector representation [22] that enables efficient searches and contextual retrieval of information.

For each design rule provided, our approach performs a search on the data to retrieve parts of the documentation relevant to testing the aspects described in each rule. The retrieved documentation is then passed to the model, along with the rule and instructions for generating the test code.

To generate the responses, we use the CodeStral model from Mistral AI [1], which is specialized in code-related tasks. CodeStral has demonstrated strong performance in benchmarks involving both natural language and programming tasks. Unlike general-purpose models, it is specifically trained to handle code generation and understanding across multiple programming languages. This specialization is particularly relevant for our study, as the proposed approach relies on accurately interpreting natural language design rules and translating them into valid and meaningful test code.

3.2.3 *Evaluation*

The tests generated were executed and analyzed from multiple perspectives. To address RQ_1, we verified whether each generated code snippet compiled successfully and whether its assertions passed during execution. To address RQ_2, we conducted a qualitative analysis of each test, evaluating its alignment with the corresponding design rule, the completeness of the validation performed, and the correct usage of the Design Wizard library. Finally, to address RQ_3, we examined the tests that failed due to errors, estimating the effort required to correct them and transform them into valid rule checkers.

4 Preliminary Results

Each design rule in our dataset led to one design test, totaling 30 design tests across 6 projects (each implementing a different design pattern). To evaluate our approach, we conducted a manual inspection of the generated tests based on criteria that address our research questions. We examined the generated code according to the following aspects:

[2] https://github.com/clarck/DesignPatterns/tree/master/StrategyPattern

[3] https://github.com/joaoarthurbm/designwizard/tree/master/doc

1. **Compilation**: A successful test case in this aspect is one that compiles without errors using the Java compiler.
2. **Passing Asserts**: Here, we analyzed whether the test cases contain assertions that evaluate to true and do not raise runtime errors.
3. **Rule Compliance**: In this qualitative assessment, we examined whether the generated code includes the necessary checks to enforce the specified design rule.

Table 2 presents an overview of the performance across all evaluation criteria, considering the full dataset.

Criterion	Passing Cases	Percentage
Successful Compilation	29/30	96.67%
Passing Asserts	19/30	63.33%
Rule Compliance	23/30	76.67%

Table 2: Overall performance summary of the test cases

For each aspect, we analyzed the causes that led the test cases to fail the corresponding criterion. We also examined whether these failures were associated with the rule difficulty levels discussed in Section 3.2.1, and estimated the effort required to correct problematic tests. Below, we present our findings for each research question.

Listing 4 shows a generated test for the rule "The factory class must not store created instances," related to the Simple Factory Pattern[4]. The test uses DesignWizard to inspect fields of the factory class and verifies that none are of type Product.

This example illustrates the potential of our approach to produce executable design conformance tests from natural language rules, though it also reveals room for improvement — for instance, the type check could compare actual types rather than relying on class name strings.

Listing 4: Generated rule example

```
@Test
public void testFactoryDoesNotStoreInstances()
    throws Exception {
    // Always instantiate DesignWizard exactly
        like this
    DesignWizard dw = new DesignWizard("bin");

    // Get the factory class
    ClassNode factoryClass = dw.getClass("com.
        cnblog.clarck.ProductFactory");

    // Get all fields of the factory class
    Set<FieldNode> fields = factoryClass.
        getAllFields();

    // Assert that there are no fields that store
        instances of created objects
    for (FieldNode field : fields) {
        ClassNode fieldType = field.getType();
        assertFalse(fieldType.getClassName().
            startsWith("com.cnblog.clarck.Product"
            ));
    }
}
```

[4]https://github.com/clarck/DesignPatterns/tree/master/SimpleFactoryPattern

4.1 RQ$_1$: Can LLMs effectively generate design tests?

To answer this question, we executed the generated design tests to identify compilation or runtime errors and analyzed the underlying causes when such issues occurred. Overall, the generated code was largely valid: approximately 96% of the test cases compiled successfully, and 63% passed all runtime assertions. Most issues were minor and could be resolved with small corrections.

The first evaluation criterion — successful compilation — was met in 96.67% of the cases. Only one test failed to compile due to a small typo: the method `getSubClasses()` from the Design Wizard library was mistakenly written with a lowercase "c" as `getSubclasses()`. Despite this minor error, the code's logic correctly reflected the intended rule.

The second criterion was assertion correctness. Out of the 30 test cases, 19 (63.33%) passed all assertions. The 11 failing tests were manually analyzed and their causes grouped into three categories: Rule Mismatch, Library Misuse, and Programming Language Errors.

4.1.1 Rule Mismatch (5 cases) These failures stemmed from the model misunderstanding the rule or using inadequate mechanisms to verify it. For example, in response to the rule *"Use composition instead of inheritance in the Adapter to wrap the adaptee, keeping the two classes loosely coupled"*, the model incorrectly verified whether the `Adapter` extends the `Target` class, rather than focusing on the relationship between `Adapter` and `Adaptee` (Listing 5).

Listing 5: Generated rule example

```
// Check if Adapter extends Target
assertFalse(adapterClass.extendsClass(targetClass)
    );
// Check if Adapter has a field of type Adaptee
Set<FieldNode> fields = adapterClass.
    getDeclaredFields();
boolean hasAdapteeField = false;
for (FieldNode field : fields) {
    if (field.getType().equals(adapteeClass)) {
        hasAdapteeField = true;
        break;
    }
}
assertTrue(hasAdapteeField);
```

4.1.2 Library Misuse (4 cases) These issues involved incorrect usage of the Design Wizard library, such as invalid arguments. In Listing 6, for instance, the test aims to verify the existence of a method named `algorithmInterface`. However, it fails because the `getShortName()` method returns the method name including parentheses (e.g., `algorithmInterface()`), which prevents exact string matching.

Listing 6: Library misuse example

```
MethodNode algorithmMethod = abstractMethods.
    iterator().next();
assertEquals("algorithmInterface", algorithmMethod
    .getShortName());
```

4.1.3 Mistakes in Programming Language Use (2 cases) These failures were due to misunderstandings of basic Java or object-oriented concepts. One case involved the rule *"The Adapter should expose only the methods defined in the Target class."* Here, the `Adapter` correctly extended the `Target` class and overrode its methods, but the generated test incorrectly assumed that method sets should be exactly equal, ignoring the semantics of method overriding (Listing 7).

Listing 7: Mistake in Programming Language Use example

```
// Get the methods defined in the Adapter class
Set<MethodNode> adapterMethods = adapterClass.
    getDeclaredMethods();
// Get the methods defined in the Target class
Set<MethodNode> targetMethods = targetClass.
    getDeclaredMethods();
// Get the methods defined in the Adaptee class
Set<MethodNode> adapteeMethods = adapteeClass.
    getDeclaredMethods();
// Check that the Adapter class only exposes
    methods defined in the Target class
for (MethodNode method : adapterMethods) {
    assertTrue(targetMethods.contains(method));
    assertFalse(adapteeMethods.contains(method));
```

Answer to RQ$_1$

LLMs showed strong potential in generating executable design tests: 96.67% compiled successfully, and 63.33% passed all runtime assertions. Of the 11 failures, some were minor (e.g., typos, library misuse), while others stemmed from misinterpreted design rules. Overall, the results indicate that LLMs can effectively generate design tests, although some manual refinement may be needed to ensure correctness.

4.2 RQ$_2$: How well do these generated tests adhere to the corresponding design rules?

To answer this question, we manually analyzed the generated test cases to verify whether they effectively tested the intended meaning of each design rule. We based our analysis on the following criteria: a test case was faithful to the rule if (i) it derived checks based on the correct relationships and entities in the system, and (ii) it was complete with respect to the aspects covered by the rule.

We found that 23 out of 30 test cases (76.67%) were faithful to their corresponding rules. Among the 7 test cases that did not meet the criteria, 5 had already been discussed as causes of assertion errors in Section 4.1.1. The remaining 2 resembled the first example shown in the Section, where the test deviated from the rule's intent. Although the assertions passed, these tests ended up verifying different conditions unrelated to the original rule.

We analyzed the correlation between these mismatches and the difficulty levels of the rules, as defined in section 3. Of the 7 problematic test cases, 5 were associated with rules classified as difficulty level 2 or 3. This indicates that there is room for improvement in the model's ability to interpret more complex rules. However, for lower difficulty levels, the generated tests tend to be more reliable. Figure 2 compares the total number of rules per difficulty level with the number of test cases that presented errors.

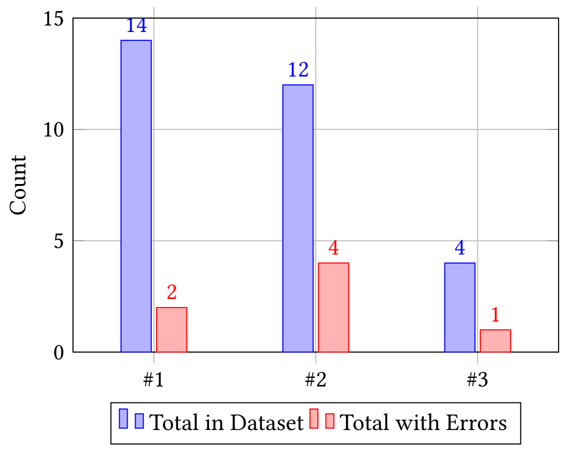

Figure 2: Comparison between total number of rules and number of errors by difficulty level.

It is evident that the proportion of errors is significantly higher for rules classified as difficulty levels 2 and 3. This reinforces the hypothesis that more complex design rules pose greater challenges for automatic test generation and require more robust strategies to ensure proper interpretation.

Answer to RQ$_2$

Most LLM-generated design tests (76.67%) correctly followed the intended rules. Of the 7 that did not, 5 also failed runtime assertions (as discussed in RQ$_1$), and 2 passed assertions but deviated from the intended rule. Notably, 5 of these 7 involved rules of difficulty level 2 or 3, suggesting LLMs handle simpler rules well but struggle with more complex ones, indicating a need for better guidance or post-processing.

4.3 RQ$_3$: What is the effort to fix the design tests that have problems?

Finally, we analyzed the design test code to assess the effort required to fix the problematic cases. Our analysis showed that, in most cases, only minor edits were needed to make the tests ideal for verifying the intended rules. Therefore, even the tests that initially failed can still be considered valuable contributions to system conformance checking, as the model generated the bulk of the code, and a human developer would only need to make small adjustments.

We reviewed all 15 test cases that exhibited issues (whether due to compilation errors, runtime errors, or rule mismatches) and identified the changes required to fix each one. Of these, 9 could be corrected with at most four lines of code—typically involving a method name change or the addition of an argument. For example, to execute the code in Listing 8, one would simply need to replace the calls to `getCalleeClasses` with `getCallerClasses`.

The remaining 6 problematic cases required more substantial changes, involving restructuring the test logic.

In summary, 24 out of the 30 generated test cases either worked correctly or required only minor adjustments to faithfully verify

Listing 8: Library misuse example

```
// Get the classes that call the subsystems
Set<ClassNode> subSystemOneCallerClasses =
    subSystemOne.getCallerClasses();
Set<ClassNode> subSystemTwoCallerClasses =
    subSystemTwo.getCallerClasses();
Set<ClassNode> subSystemThreeCallerClasses =
    subSystemThree.getCallerClasses();
Set<ClassNode> subSystemFourCallerClasses =
    subSystemFour.getCallerClasses();
```

their corresponding rules. This suggests that using LLMs to assist in generating design conformance tests holds promise as a supportive tool in software architecture verification.

Answer to RQ$_3$

Of the 15 flawed test cases, 9 required only minor fixes — such as renaming methods or adding arguments. This suggests that even imperfectly generated code is easily repairable and still valuable for system conformance checking.

5 Related Work

Several studies address the problem of architectural drift and erosion, and propose tools to help mitigate this challenge. Whiting et al. [21] conducted a comprehensive survey of methods and tools proposed to detect architectural degradation, highlighting how this issue has been addressed within the field of Software Architecture. visualizations that assist in understanding the detected violations.

ArchPython [7] is an architectural conformance checking tool for Python systems. It uses a JSON file with architectural rules, performs type inference, and analyzes code compliance, generating reports and visualizations of violations.

Rodríguez et al. [18] present ArchLearner, a tool that detects deviations between the expected architecture and the observed implementation. The architectural input consists of *Use Case Maps* (UCMs), which are compared to the behavior observed during the system's execution. The correspondence between UCM steps and execution events is used to generate a knowledge model, based on Markov models, that describes the expected behavior. This model is then applied to future versions to identify behavioral divergences.

Chen et al. [8] propose an approach that models both the architecture and source code as graphs, evaluating consistency between these representations through the calculation of structural similarities. Inconsistencies are detected based on the lack of correspondence between architectural entities and code elements. From an initial mapping, the method automatically infers additional mappings to assess the overall consistency of the system.

Miño et al. [14] propose an innovative approach to detecting design flaws by incorporating generative language models (LLMs) into the process. Unlike previous approaches, this study does not rely on a reference architecture or formal documentation, and the goal is not to check conformance with a specific design. The focus is on identifying antipatterns directly in the source code, through automated analysis coupled with predefined antipattern definitions. The model identifies potential design flaws and suggests corrections to mitigate them.

Although there is a significant body of work proposing solutions to the problem of architectural drift, most approaches rely on the existence of formal and up-to-date documentation of the intended design. Approaches that leverage informal and unstructured discussions about design decisions for conformance checking are still scarce. Despite the ability of LLMs to process natural language and unstructured data, their use in transforming such discussions into formal verification artifacts remains underexplored. This work aims to bridge that gap by investigating the use of LLMs to generate design verification tests from informal descriptions, promoting traceability between design decisions and implementation without relying on formal documentation.

6 Conclusions and Future Work

This paper presented an initial study on the use of Large Language Models (LLMs) for generating design tests aimed at verifying architectural conformance. Preliminary results indicate that our approach has the potential to translate the intent behind design rules into executable verification code, although challenges remain — especially with more complex rules.

A key implication of this study is the possibility of automating architectural conformance checking based on informal developer discussions, without the need for formally documented rules. This could significantly simplify the process of maintaining architectural integrity throughout a system's life cycle, particularly in agile or rapidly evolving projects.

We acknowledge some threats to validity in this work. These include potential bias in the manual classification of rule difficulty levels and in the subjective analysis of whether a test correctly implements a rule. Moreover, since our evaluation was based on synthetic rules, it may not fully capture the nuances of real-world developer communication. Future studies involving multiple annotators and rules extracted from actual project histories could help mitigate these limitations.

As future work, we plan to extend our evaluation to larger, real-world systems using design rules derived from practical development scenarios. We also intend to experiment with alternative libraries and tools for rule formalization beyond Design Wizard, as well as evaluate the performance of different LLMs in test generation—since the choice of model may significantly impact the quality and accuracy of the generated tests.

Finally, we aim to consolidate our findings into a more comprehensive solution by formalizing them into a technique or tool to support continuous architectural conformance checking. Such a system could extract design rules from communication channels (e.g., code reviews, issue discussions), generate verification tests automatically, and integrate with CI/CD pipelines or IDEs. This would enable development teams to detect architectural violations early, with minimal manual effort, and promote sustainable architectural practices over time.

ARTIFACT AVAILABILITY

All code and data are publicly available at https://doi.org/10.5281/zenodo.16994021 [13]. Instructions for reproduction are provided in the README file.

REFERENCES

[1] Mistral AI. 2024. Codestral. Mistral AI. https://mistral.ai/news/codestral Accessed: 23 Mar. 2025.

[2] Emilie Anthony, Astrid Berntsson, Tiziano Santilli, and Rebekka Wohlrab. 2024. We're Drifting Apart: Architectural Drift from the Developers' Perspective. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*. IEEE, 101–111.

[3] Len Bass, Paul Clements, and Rick Kazman. 2012. *Software Architecture in Practice* (3rd ed.). Addison-Wesley Professional.

[4] David Baum, Jens Dietrich, Craig Anslow, and Richard Müller. 2018. Visualizing Design Erosion: How Big Balls of Mud are Made. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*. 122–126. doi:10.1109/VISSOFT.2018.00022

[5] Joao Brunet, Dalton Guerrero, and Jorge Figueiredo. 2009. Design tests: An approach to programmatically check your code against design rules. In *2009 31st International Conference on Software Engineering-Companion Volume*. IEEE, 255–258.

[6] João Brunet, Gail C Murphy, Ricardo Terra, Jorge Figueiredo, and Dalton Serey. 2014. Do developers discuss design?. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 340–343.

[7] Andrea Caracciolo, Mircea Filip Lungu, and Oscar Nierstrasz. 2015. A unified approach to architecture conformance checking. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*. IEEE, 41–50.

[8] Fangwei Chen, Li Zhang, and Xiaoli Lian. 2020. An improved mapping method for automated consistency check between software architecture and source code. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 60–71.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[10] Eduardo F de Lima and Ricardo Terra. 2020. ArchPython: architecture conformance checking for Python systems. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*. 772–777.

[11] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[12] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[13] Andrielly Lucena. 2025. *andriellyll/design-test-generation: Reproducibility Package – Language Models as Architectural Gatekeepers*. doi:10.5281/zenodo.16994021

[14] Jorge Miño, Roberto Andrade, Jenny Torres, and Kharol Chicaiza. 2024. Leveraging Generative Artificial Intelligence for Software Antipattern Detection. In *International Conference on Information Management*. Springer, 138–149.

[15] OpenAI. 2024. GPT-4o. OpenAI. https://openai.com/research/gpt-4o Accessed: 21 Mar. 2025.

[16] PostgreSQL Global Development Group. 2024. *PostgreSQL Documentation*. https://www.postgresql.org/docs/ Acessado: 2025-03-23.

[17] Mengchao Ren. 2024. Advancements and Applications of Large Language Models in Natural Language Processing: A Comprehensive Review. *Applied and Computational Engineering* 97 (2024), 55–63.

[18] Guillermo Rodriguez, Marcelo Armentano, Álvaro Soria, and Emilio Corengia. 2020. Evaluation of Markov Models for Architecture Conformance Checking. *IEEE Latin America Transactions* 18, 01 (2020), 43–50.

[19] Daniel Gustavo San Martín Santibánez. 2021. REMEDY: architectural conformance checking for adaptive systems. (2021).

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[21] Erik Whiting and Sharon Andrews. 2020. Drift and Erosion in Software Architecture: Summary and Prevention Strategies. In *Proceedings of the 2020 the 4th International Conference on Information System and Data Mining* (Hawaii, HI, USA) *(ICISDM '20)*. Association for Computing Machinery, New York, NY, USA, 132–138. doi:10.1145/3404663.3404665

[22] Haozhou Zhao. 2023. pgvector: Open-source vector similarity search for Postgres. https://github.com/pgvector/pgvector. Accessed: 17 May 2025.