# LLM-Based Recommender Systems for Violation Resolutions in Continuous Architectural Conformance

1st Riccardo Rubei
*University of L'Aquila*
L'Aquila, Italy
riccardo.rubei@univaq.it

2nd Amleto Di Salle
*Gran Sasso Science Institute*
L'Aquila, Italy
amleto.disalle@gssi.it

3rd Alessio Bucaioni
*Mälardalen University*
Västerås, Sweden
alessio.bucaioni@mdu.se

***Abstract*—Software architectures are fundamental to the development, evolution, and quality of software-intensive systems. Architectures rarely exist in isolation, but instead adhere to overarching structures such as architectural patterns and styles, frameworks, software product line architectures, and reference architectures. To fully leverage the benefits of these structures, conformance between them is essential, enhancing interoperability, reducing costs through reusability, mitigating project risks, and facilitating the adoption of best practices. In our previous work, we introduced the concept of continuous conformance and focused on detecting architectural violations using a model-driven engineering approach.**

**In this paper, we extend our previous work by proposing a large language model-based recommender system into the model-driven tool to suggest resolutions for architectural violations. Leveraging large language models, we reduce the accidental complexity of model-driven techniques by combining the reasoning capabilities of large language models with the formalization of architectures as (meta)models. We evaluate the success rate using two large language models and architectures from the IoT domain, including one reference architecture and four software architectures that we manually mutate in 16 faulty architectures. The results demonstrate the system's effectiveness in providing intelligent, context-aware recommendations for restoring architectural conformance.**

***Index Terms*—Large language models, recommender systems, model-driven engineering, architectural conformance**

## I. Introduction

Software architectures (SAs) are widely recognized as the backbones of software-intensive systems, playing a vital role in determining their quality, development, and evolution [1]. SAs are rarely developed nor used in isolation; instead, they follow overarching structures and principles to guide their standardization and evolution within specific domains or organizations [2]. Examples include architectural patterns and styles, architectural frameworks, software product line architectures, and reference architectures (RAs). For software systems to fully leverage the benefits of these architectural structures, a degree of conformance between them is necessary. Such conformance enhances interoperability, reduces costs through re-usability, decreases project risks, improves communication among stakeholders, and facilitates the adoption of best practices. By aligning architectural structures, organizations can maximize the effectiveness of their software systems and ensure their architectures remain both robust and adaptive to change.

In our previous work, we introduced the concept of *continuous conformance*, which measures the optimal alignment between any two architectural structures at a given time [3]. The continuous conformance concept enables multi-level, incremental, non-blocking checking and restoration tasks. Furthermore, it allows for validating partial architectures without obstructing the overall design process. In addition, we operationalized this concept by formalizing SAs as (meta)models and by developing a Model-Driven Engineering (MDE) tool, named AssistRA, that supports continuous conformance by detecting architectural violations [3]. However, our work did not address the automatic resolution of these detected violations, leaving this a promising future research direction. Specifically, we envisioned leveraging Recommender Systems (RS) to automate and enhance the resolution process.

In this paper, we extend our previous work by proposing a recommender system into the MDE tool to suggest resolutions for architectural violations. By building on current trends in utilizing Large Language Models (LLMs) to reduce the accidental complexity associated with MDE techniques, we leverage the reasoning capabilities of LLMs in combination with the formalization of architectures as (meta)models to deliver intelligent, context-aware recommendations for restoring architectural conformance. Specifically, we propose an LLM-based recommender system to address architectural violations and evaluate its preliminary applicability and efficacy using illustrative examples from the IoT domain, including one RA and four SAs. Based on the RA and SAs, we manually create 16 faulty SAs, which we use as input for two LLMs: Gemini 1.5 and ChatGPT-4o. Results indicate that the LLM-based recommender system exhibits promising performance, achieving at least 60% correct suggestions in certain configurations.

This work has been partially funded by (a) the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem, grant ECS00000041 - VITALITY – CUP: D13C21000430001, (b) PRIN 2020 program grant n. 2020W3A5FY - EMELIOT, (c) Italian "PRIN 2022" TReX-SE grant n. 2022LKJWHC,(d) the MUR (Italy) Department of Excellence 2023 - 2027, (e) the European HORIZON-KDT-JU research project MATISSE "Model-based engineering of Digital Twins for early verification and validation of Industrial Systems", HORIZON-KDT-JU-2023-2-RIA, Proposal number: 101140216-2, KDT232RIA_00017, (f) the Swedish Knowledge Foundation through the MoDEV project (20200234), and (g) the Sweden's innovation agency Vinnova through the project iSecure (202301899)

The remainder of this paper is organized as follows. Section II provides the background concepts. Section III describes the proposed LLM-based recommender system. Section IV presents the experimental results, illustrating the system's applicability and efficacy. Section V discusses existing works related to our approach. Section VI concludes the paper with final remarks and outlines potential future work.

## II. Background

Several definitions of RAs have been provided in the last three decades [2], [4]. All these definitions highlight that a RA is a *template* used when specifying a concrete SA. In our work, we leverage the definition of the well-known Systematic Mapping Study (SMS) study on Reference Architecture by Lina Garcés et al. [5]. In particular, the authors define an RA as *an abstraction of software elements, together with the main responsibilities and interactions of such elements, capturing the essentials of existing software systems in a domain and serving as a guide for the architectural design of new software systems (or versions of them) in the domain*. The study also states that 85% of the identified primary studies adopted an ad-hoc approach to design the 162 RAs, even though an architecture language (AL) can define RAs [5].

Different ALs, such as AADL [1], ArchiMate [2], and C4 Model [3], have been proposed over the years. Typically, ALs support aspects and concerns such as structural, behavioural, functional, and programmatic. However, they do not allow defining a RA as a first-class citizen to be used as a building block during the definition of SA. In other words, a SA that conforms to a RA can not be specified through these ALs. Hence, their conformance must be manually verified and validated by the software architect.

To overcome these challenges, we defined a model-driven approach named MORE to model RAs [6]. Our approach automatically checks the conformance of a RA with defined guidelines and rules, e.g., styles, and of a concrete SA concerning a RA. The approach is binary in that the conformance is satisfied or not. Recently, we extended the MORE approach by introducing the *continuous* concept. In particular, we defined continuous conformance as a distance function that estimates the degree to which an SA conforms to a specific RA. The approach can also be applied to an RA conforming to an architectural style, thus enabling multi-level, incremental, and non-blocking conformance checking. We also developed an assistive modeling tool that implements the distance function, named AssistRA. The tool helps architects create an SA aligned with a specific RA or more abstract architectures by automatically detecting and restoring misalignments.

In the following, we use the defined language in our assistive modeling tool to describe an RA in the Internet of Things (IoT) domain, and the FIWARE SA borrowed fromt the work by Guth et al. studies [7], [8]. Figure 1 shows the RA from the IoT domain and comprises five layers containing seven components, where the bottom contains the Sensors and Actuators components. The IoT RA also employs the Publish-Subscribe architectural style [9]. The Device intermediates between the hardware and the counterpart software, allowing a driver to access a sensor or actuator. A Gateway is used when a Device can not be attached to other systems. A Gateway converts various protocols, communication technologies, and payload formats. The last layer, before the Application, is the IoT Integration Middleware (IoTIM). It is a broker for Sensors, Actuators, Devices, and Applications. It receives data from a Device, processes data, and sends data to the connected Applications. A Device can communicate directly with the IoTIM if it supports a suitable communication technology.

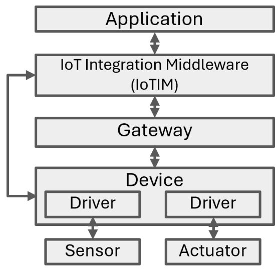

Fig. 1. IoT reference architecture

Listing 1 represents an excerpt of the same RA using our language defined in [3]. The language is based on Flexmi, which defines domain-specific languages using a JSON-style notation [10] and employs the components and connectors view to represent architectures [2]. In particular, lines 2–4 describe the components Application, IoTIM, and Gateway, which act as a subscriber, broker, and publisher in the Publish-Subscribe style, respectively. Lines 5–7 represent two-way connectors among Application, IoTIM, and Gateway.

```
1 ArchitecturalModel: {
2   Component: {name: Application, implements:
    Subscriber},
3   Component: {name: IoTIM, implement: Broker},
4   Component: {name: Gateway, implements:
    Publisher},
5   Connector: {source: Application, target: IoTIM
    , twoWay},
6   Connector: {source: Gateway, target: IoTIM,
    twoWay },
7   Connector: {source: Gateway, target: Device,
    twoWay},
8 }
```

Listing 1. Internet of Things reference architecture

Listing 2 shows an excerpt of the FIWARE architecture conforming to the IoT RA.

```
1 ArchitectureModel: {
2  Component: {name: MyApp, implements:
   Application},
3  Component: {name: Data Context Broker,
   implements: IoTIM},
4  Component: {name: Device1, implements: [Device
   , Sensor, Actuator]},
5  Connector: {source: MyApp, target: Device1,
   twoWay}
6 }
```

Listing 2. FIWARE software architecture

The `implements` keyword is used to specify a component in the architecture that implements the related component in the RA. For example, line 2 expresses the MyApp component implementing the Application component defined in the RA.

[1]http://www.openaadl.org/
[2]https://www.opengroup.org/archimate-forum/archimate-overview
[3]https://c4model.com/

The SA also contains a violation concerning the RA. In particular, Device1 is connected to the MyApp component, and the IoT RA only considers connectors between Application and IoTIM components (line 5).

Building an SA is often a complex and time-consuming process. A challenge further intensified when the SA must conform to an RA [11]. Therefore, automatic support during the design of an SA by the software architect should be desirable.

## III. Proposed Approach

Figure 2 illustrates our proposed approach for assisting architects in resolving inconsistencies SAs. During the architecting and refinement stages, any violations identified by our model-driven assistive tool are reported to the architect (①). The tool's validation service generates a tuple on the dashboard that specifies the component responsible for violating a particular architectural constraint. This tuple, along with the RA and the SA under development, is used to construct a query prompt (②). Next, the LLM is queried to address the specific violation identified (③). The LLM processes the input and generates a potential resolution, which is then presented to the architect as a proposed solution (④).

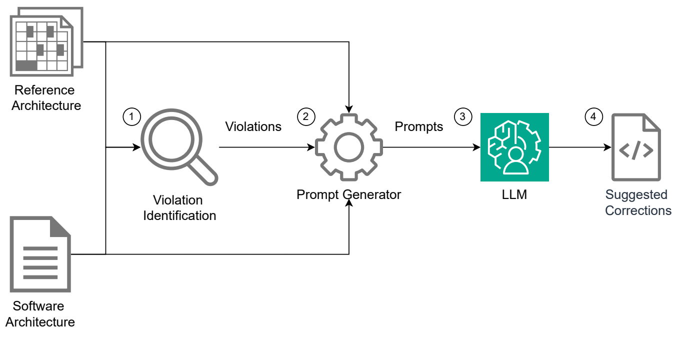

Fig. 2. Architecture of the proposed approach

### A. Violation Identification

We base the violation identification on our previously developed model-driven tool, AssistRA [3], designed to support architects in defining and evolving architectures. This tool operationalizes the concept of continuous conformance to verify the alignment between an RA and an SA under development. AssistRA's validation service offers detailed insights into detected violations, making it a valuable resource for extracting relevant information. Specifically, we leverage this capability to classify a subset of possible violations and identify the actors involved, as detailed below. It is worth noting that the list of violations provided is not exhaustive but includes a few representative examples. In general, these violations depend on several factors, such as the specific style of the RA and any additional constraints applied and cannot be applied to a generic SA.

*a) Link Violation:* RAs explicitly define which components (source) can connect to other components (target). A Link Violation occurs when a connection in the SA is established between two components not permitted by the RA.

*b) Component Omission Violation:* This type of violation arises when a connection is defined without specifying the target component, leaving it incomplete.

*c) Mandatory Component Violation:* RAs can include additional user-defined constraints. For example, certain components may be mandatory. This type of violation arises when a mandatory component in the RA is not defined in the SA.

In addition to the violations above, architectures under development may simultaneously exhibit combinations of multiple violations. As an example, and for simplicity, we consider the conjunction of a Link Violation and a Component Omission Violation.

*d) Link and Component Omission Violation:* This type of violation arises when architectures include an illegal connection alongside a connection where the target component's specification is omitted.

### B. Prompt Generation

This component is responsible for generating the prompt used to query the LLM. The core idea behind the prompt generator is to create a query adaptable to any generic LLM. To achieve this, the component utilizes the RA, the SA under development, the violation tuple, and the target LLM. For this experiment, we define four distinct prompt skeletons corresponding to each type of violation examined.

*a) Link Violation Prompt:* The prompt is constructed using the source component and the incorrect target component, emphasizing that a direct connection between them is not allowed. No additional information is provided to the LLM to ensure a generic and unbiased query. Listing 3 illustrates an example of a prompt designed to address a Link Violation.

```
Hi! I'm giving you an example of a Reference
Architecture (RA), and a concrete Software
Architecture (SA) based on the RA. The SA
contains an error, the component "MyApp" cannot
be directly connected to the component "Device1
". Can you solve this violation? Thank you

"Reference Architecture": "ArchitectureModel:
{...}",

"Software Architecture": "ArchitectureModel:
{...}"
```

Listing 3. Link violation prompt

*b) Component Omission Violation:* In the case of a Component Omission Violation, the SA contains a component that lacks a connection to another component. This violation occurs when the connector does not specify the target of the connection. Listing 4 presents an example of a prompt designed to address this type of violation. For the sake of conciseness, we omitted the RA and SA definitions from this and the following example listings.

```
Hi! I'm giving you an example of a Reference
Architecture (RA), and a concrete Software
Architecture (SA) based on the RA. The SA
contains an error, the component "MyApp" is not
connected to any component. Can you solve this
violation? Thank you
```

Listing 4. Component omission prompt

*c) Mandatory Component Violation:* A RA may have the concept of mandatory component. In the prompt, we explicitly indicate the component of a SA that fails to implement any required RA component. Since some components may have multiple implementation alternatives, we omit details about these options, delegating the decision-making to the LLM's reasoning capabilities. An example of such a prompt is provided in Listing 5.

```
Hi! I'm giving you an example of a Reference
Architecture (RA), and a concrete Software
Architecture (SA) based on the RA. The SA
contains an error, the component "MyApp" is not
implementing anything. Can you solve this
violation? Thank you
```

Listing 5. Mandatory component prompt

*d) Link and Component Omission Violation:* We define the prompt by combining the Link Violation and Component Omission scenarios. In this case, the architecture violates the RS by including both an inadmissible connection and a missing connection. The prompt is constructed by integrating elements from the prompts defined in Listings 3 and 4. An example of this combined prompt is shown in Listing 6.

```
Hi! I'm giving you an example of a Reference
Architecture (RA), and a concrete Software
Architecture (SA) based on the RA. The SA
contains an error, the component "MyApp" cannot
be directly connected to the component Device1
and the component "IoT Device Management" is not
 connected to any component. Can you solve this
violation? Thank you
```

Listing 6. Link and component omission prompt

### C. Suggested Correction

Once the prompt is received, the LLM generates a response to fix the architecture with violations. The output consists of a textual document that includes a new version of the architecture along with a succinct description of the modifications. This format offers two key advantages: first, the new architecture is immediately ready for evaluation by the architect; second, the textual description provides developers with a clear understanding of the rationale behind the modifications and the components affected. Once the architect approves the suggested changes, the updated architecture is integrated, allowing the development process to proceed seamlessly. A concrete example of the produced architecture is presented in Section IV. It is worth noting that in this exploratory study, we do not structure the prompts to produce a strictly predefined output format. Defining a practical output format and optimizing the presentation of results to the architect are immediate steps we plan to pursue in future work.

## IV. Experiment

To evaluate the efficacy of the proposed methodology, we established a three-step evaluation framework. First, a set of SAs is deliberately mutated to introduce simulated violations. Second, the architectural recommendations generated by the LLMs are manually compared with the original, unmutated architectures. Finally, the effectiveness of the approach is quantitatively assessed by calculating the proportion of LLM-generated suggestions that are deemed acceptable.

### A. Architecture Mutation

In our experiment, we consider a set of four SAs and one RA taken from the work by Guth et al. [7]. In particular, we utilize the IoT RA, AWS IoT, Azure IoT Hub, FIWARE, and SmartThings, as they provide well-defined components and connection specifications. Based on the violation identified by AssistRA, we define a set of mutations applied to the original SA to test the model's fixing capability. We describe the specific mutations as follows:

*a) Link Violation:* We mutate the SA by substituting the target of a connection with a non-admissible component.

*b) Component Omission Violation:* We mutate the SA by removing the target of a connection.

*c) Mandatory Component Violation:* We remove the implementation definition from the architecture.

*d) Link and Component Violation:* We simultaneously employ the link and component omission violation mutation to the same SA. Therefore, we evaluate two suggestions for this violation.

Considering the above mutations for each SA, we reached a test-bed of 16 total SAs.

### B. Evaluation

To evaluate the capability of the proposed approach in resolving inconsistencies, we assess the correctness of the solutions provided by the LLM by manually comparing the fixed architecture with the original one. Since the architecture under development can be completed in multiple ways—for instance, a connection may be valid with different target components—we define three distinct degrees of correctness to guide our evaluation.

*a) Correct Match:* A correct match occurs when the LLM suggests an admissible solution for the architecture. For instance, if a component can be legally connected to multiple components, suggesting any one of those valid components constitutes a correct match.

*b) Wrong Match:* A wrong match is identified when the LLM's proposed solution introduces another violation. For example, suggesting a connection to a component that violates the rules defined by the RA.

*c) Undesired Action:* This category includes cases where the LLM suggests unnecessary or inappropriate actions. Examples include adding, removing, or modifying elements unrelated to the violation, or offering trivial solutions, such as simply removing the problematic element.

We use these three degrees of correctness to calculate the success rate as the proportion of cases where the model produces a Correct Match out of all evaluated scenarios. This metric provides a quantitative measure of the model's ability to assist architects effectively while minimizing undesired and wrong actions.

### C. Example of Suggested Correction

Listing 7 reports an admissible solution to a link violation as described in Section II.

```
"The rest of the architecture is omitted"
Connector: {source: MyApp, target: IoT Broker,
twoWay},
Connector: {source: IoT Broker, target: Device1,
 twoWay}
```

Listing 7. Suggested Correction

In this particular example, we omit the rest of the architecture for the sake of conciseness. The suggested solution is a modification of the Connector MyApp - Device1. The LLM propose to first link MyApp to the IoT Broker and then connect the IoT Broker to Device1. We can notice that both the connections are legal, as described in the RA in Listing 1. The LLMs suggested a copy of the mutated architecture with some modifications. Generally, modifications pertain only to the violations. In rare cases, the LLMs modified the architecture by applying and reporting further optimization that was not explicitly required. In our experiment, we explicitly handled these undesired actions, as Section IV explains. The corrected architecture is then proposed to the architect who can accept the suggestions and continue the development. At this stage of the study, we do not develop this presentation layer as it requires to be embedded in the tool.

### *D. Results*

Table I reports the results of the experiments. We describe the amount of *Correct Matches*, *Undesired Actions*, and *Wrong Matches* calculated on the answer produced by the LLMs. The results reported for a particular architecture represent the cumulative success rate for that LLMs across all the mutations. For example, to fix the violations for the architecture AWS, Gemini got five correct matches. This means that Gemini correctly solved the *Link Violation*, *Component Omission*, *Mandatory Component* and *Link and Component Omission*. We observed a number of positive results. Notably, both Gemini and ChatGPT achieved at least 60% correct matches, indicating that the models are capable of providing valuable suggestions in most cases. To ensure a fair evaluation, we did not set or adjust the models' temperature values during testing. Finally, we observed a generally favorable trend regarding incorrect matches, with only one instance where a model proposed more than one erroneous fix. Summing up the correct matches for Gemini and ChatGpt, we obtain a total of 90% and 70% of success rate, respectively. Gemini never suggested an undesired action, whereas ChatGpt proposed a total of 5 over 16. Finally, concerning the wrong recommendations, Gemini suggested two wrong fixes over 16, while ChatGpt performed slightly worse, suggesting 25% of erroneous corrections.

TABLE I
SUMMARY OF THE EXPERIMENTAL RESULTS

|  | **Correct Match** |  | **Undesired Action** |  | **Wrong Match** |  |
|---|---|---|---|---|---|---|
| **SAs** | **Gemini** | **ChatGpt** | **Gemini** | **ChatGpt** | **Gemini** | **ChatGpt** |
| AWS | 5 (100%) | 4 (80%) | 0 | 0 | 0 | 1 (20%) |
| Azure | 4 (80%) | 3 (60%) | 0 | 3 (60%) | 1 (20%) | 1 (20%) |
| FIWARE | 5 (100%) | 3 (60%) | 0 | 0 | 0 | 2 (40%) |
| Smart Things | 4 (80%) | 4 (80%) | 0 | 2 (40%) | 1 (20%) | 1 (20%) |

Table II shows a detailed description of the different fixes proposed by the model, focusing on the particular violation.

Each row reports the status of the suggestions for the entire set of violations. A ✓ indicates a positive evaluation of the model's fix, a ✗ represents an incorrect suggestion, and a ✻ denotes cases involving undesired actions. This perspective provides a clearer understanding of the LLMs' capability to address specific violations. For the *Link Violation*, Gemini made one error, while ChatGPT produced no mistakes but generated undesired actions on two occasions. Interestingly, both Gemini and ChatGPT failed to provide the correct answer for the Azure architecture. For the *Component Omission* violation, both Gemini and ChatGPT suggested correct fixes. However, ChatGPT proposed additional modifications for Azure and SmartThings, resulting in undesired actions. A similar trend is observed for the *Mandatory Component* violation, where Gemini achieved 100% correct matches, whereas ChatGPT produced two incorrect recommendations across two architectures. Finally, for the *Link and Component Omission* violation, Gemini successfully solved seven out of eight violations, while ChatGPT produced two incorrect and two undesired suggestions.

### *E. Discussion*

Gemini tends to be less creative in its approach and avoids performing undesired actions. In contrast, ChatGPT exhibits greater independence, occasionally performing optimization operations not explicitly requested in the prompt. For instance, in two cases, ChatGPT introduced superfluous modifications to the solution. Both models found *Component Omission* violations relatively straightforward to resolve, correctly identifying the target component in broken connections. However, selecting the most appropriate target among multiple valid options ultimately depends on the architect's decision, highlighting an area for future exploration. Interestingly, the models performed worse when addressing violations in the Azure and SmartThings architectures, suggesting limitations in their ability to comprehend more complex or unique architectural styles. For example, in Azure, Gemini failed to address a *Link Violation*, while ChatGPT proposed an undesired solution by simply removing the problematic connection. Similarly, in SmartThings, ChatGPT introduced unnecessary optimizations, and both models provided incorrect solutions for *Component* violations in *Link and Component Omission* scenarios. In contrast, the AWS and FIWARE architectures yielded better results. On AWS, ChatGPT produced a single incorrect solution, while FIWARE exhibited similar outcomes, with ChatGPT generating two erroneous recommendations. These observations underscore the variability in model performance across different architectures and the need for further refinements to improve consistency and understanding of unique architectural constraints.

## V. RELATED WORK

In the following, we provide some related work employing LLM to assist software architects. To the best of our knowledge, our approach is the first to tackle the specific task of recommending fixes during the architecture development. Pace et al. [12] proposed an LLM-based tool to help architects create an Architecture Decision Record (ADR). In particular, the authors exploited the zero-shot and Retrieval Augmented Generation (RAG) prompting techniques. Similarly, Dhar et al. [13] developed a tool to assist architects in defining ADR. In this work, the authors used GPT and T5 models to generate the suggestions. Furthermore, they explored the effects of different prompting techniques (zero and one-shot) and fine-tuning. Automated Test Architecture Generation (ATAG) [14] is an LLM-based tool conceived to generate test architecture and test function names. The authors define a fine-tuned version of the BERT model to produce function names. In

TABLE II
DETAIL OF THE VIOLATIONS

| | **Link Violation** | | **Component Omission** | | **Mandatory Component** | | **Link & Component Omission** | |
|---|---|---|---|---|---|---|---|---|
| **Architecture** | **Gemini** | **ChatGpt** | **Gemini** | **ChatGpt** | **Gemini** | **ChatGpt** | **Gemini** | **ChatGpt** |
| AWS | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ and ✓ | ✓ and ✓ |
| Azure | ✗ | ✻ | ✓ | ✓ + ✻ | ✓ | ✓ | ✓ and ✓ | ✻ and ✓ |
| FIWARE | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ and ✓ | ✓ and ✗ |
| Smart Things | ✓ | ✓ + ✻ | ✓ | ✓ + ✻ | ✓ | ✓ | ✓ and ✗ | ✓ + ✻ and ✗ |
| ✓: Correct Match ✗: Wrong Match ✻: Undesired Actions | | | | | | | | |

[15], the authors explore the feasibility of applying LLM in the design and development of ensemble-based architecture. In particular, the authors tested ChatGPT 4's capability to understand the definition of ensemble architecture. In the context of model-driven engineering, Kebaili et al. [16] employed an approach similar to ours to help modellers ease the effects of model evolution. In particular, the authors studied the effects of prompting ChatGpt 3.5 to co-evolve metamodels. The evaluation of 5320 prompts demonstrates the effectiveness of ChatGpt on this task. Although conceptually similar to our approach, the context is different. We are not dealing with complexities bound to co-evolution. In a similar direction, Zhang et al. [17] propose an LLM-based tool to translate code changes from one programming language to another. Selfevolve [18] is a self-augmented code generation framework based on LLMs, particularly ChatGpt 3.5.

## VI. Conclusion and future work

This study investigates the potential of leveraging LLMs to assist software architects during the architectural process, with resolving potential violations. The evaluation involved two of the most widely used LLMs: Google's Gemini and ChatGPT. The results highlight the capability of these models to understand architectural structures and propose accurate resolutions for identified violations. In particular, Gemini and ChatGPT achieved 100% and 80% correct recommendations, respectively, for certain configurations.

For future work, we plan to extend our approach in several directions. First, we aim to fully integrate the violation resolution mechanism into the AssistRA tool to enable real-time recommendations. Additionally, we intend to explore and support advanced prompting techniques, such as Few-Shot learning and Retrieval-Augmented Generation (RAG). The ability of RAG to utilize multiple relevant examples offers significant potential for providing highly focused and context-aware recommendations.

## References

[1] P. Kruchten, H. Obbink, and J. Stafford, "The past, present, and future for software architecture," *IEEE software*, vol. 23, no. 2, pp. 22–30, 2006.

[2] L. J. Bass, P. C. Clements, and R. Kazman, *Software architecture in practice fourth edition*, ser. SEI series in software engineering. Addison-Wesley-Longman, 2021.

[3] A. Bucaioni, A. D. Salle, L. Iovino, L. Mariani, and P. Pelliccione, "Continuous conformance of software architectures," in *21st IEEE International Conference on Software Architecture, ICSA*. IEEE, 2024, pp. 112–122.

[4] E. Y. Nakagawa, M. Guessi, J. C. Maldonado, D. Feitosa, and F. Oquendo, "Consolidating a process for the design, representation, and evaluation of reference architectures," in *IEEE/IFIP Conference on Software Architecture, WICSA*, 2014, pp. 143–152.

[5] L. Garcés, S. Martínez-Fernández, L. Oliveira, P. Valle, C. Ayala, X. Franch, and E. Y. Nakagawa, "Three decades of software reference architectures: A systematic mapping study," *Journal of Systems and Software*, vol. 179, p. 111004, 2021.

[6] A. Bucaioni, A. Di Salle, L. Iovino, I. Malavolta, and P. Pelliccione, "Reference architectures modelling and compliance checking," *Softw. Syst. Model.*, vol. 22, no. 3, pp. 891–917, 2023.

[7] J. Guth, U. Breitenbücher, M. Falkenthal, P. Fremantle, O. Kopp, F. Leymann, and L. Reinfurt, "A detailed analysis of iot platform architectures: Concepts, similarities, and differences," in *Internet of Everything - Technology, Communications and Computing*. Springer, 2018, pp. 81–101.

[8] J. Guth, U. Breitenbücher, M. Falkenthal, F. Leymann, and L. Reinfurt, "Comparison of iot platform architectures: A field study based on a reference architecture," in *Cloudification of the Internet of Things*. IEEE, 2016, pp. 1–6.

[9] N. Fotiou, D. Trossen, and G. C. Polyzos, "Illustrating a publish-subscribe internet architecture," *Telecommunication Systems*, vol. 51, pp. 233–245, 2012.

[10] D. Kolovos and A. de la Vega, "Flexmi: a generic and modular textual syntax for domain-specific modelling," *Software and Systems Modeling*, vol. 22, no. 4, pp. 1197–1215, 2023.

[11] S. Angelov, P. W. P. J. Grefen, and D. Greefhorst, "A framework for analysis and design of software reference architectures," *Inf. Softw. Technol.*, vol. 54, no. 4, pp. 417–431, 2012. [Online]. Available: https://doi.org/10.1016/j.infsof.2011.11.009

[12] J. A. D. Pace, A. Tommasel, and R. Capilla, "Helping novice architects to make quality design decisions using an llm-based assistant," in *Software Architecture - 18th European Conference, ECSA*, vol. 14889. Springer, 2024, pp. 324–332.

[13] R. Dhar, K. Vaidhyanathan, and V. Varma, "Can llms generate architectural design decisions? - an exploratory empirical study," in *21st IEEE International Conference on Software Architecture, ICSA*. IEEE, 2024, pp. 79–89.

[14] G. Wang, J. Wu, H. Yang, Q. Sun, and T. Yue, "Test architecture generation by leveraging BERT and control and data flows," in *Engineering of Complex Computer Systems - 28th International Conference, ICECCS*, vol. 14784. Springer, 2024, pp. 125–145.

[15] M. Töpfer, D. Khalyeyev, T. Bures, P. Hnetynka, and F. Plášil, "How well do llms understand deeco ensemble-based component architectures," in *12th International Symposium, ISoLA 2024*, vol. 15220. Springer, 2024, pp. 208–223.

[16] Z. K. Kebaili, D. E. Khelladi, M. Acher, and O. Barais, "An empirical study on leveraging llms for metamodels and code co-evolution," *J. Object Technol.*, vol. 23, no. 3, pp. 1–14, 2024.

[17] J. Zhang, P. Nie, J. J. Li, and M. Gligoric, "Multilingual code co-evolution using large language models," in *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, 2023, pp. 695–707.

[18] S. Jiang, Y. Wang, and Y. Wang, "Selfevolve: A code evolution framework via large language models," *CoRR*, vol. abs/2306.02907, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.02907