Formal Software Architecture Rule Learning: A Comparative Investigation between Large Language Models and Inductive Techniques

Christian Schindler * and Andreas Rausch

Abstract: This paper explores the application of inferring software architecture rules from examples using Machine Learning (ML). We investigate different methods from Inductive Rule Learning and utilize Large Language Models (LLMs). Traditional manual rule specification approaches are time-consuming and error-prone, motivating the need for automated rule discovery. Leveraging a dataset of software architecture instances and a meta-model capturing implementation facts, we used inductive learning algorithms and LLMs to extract meaningful rules. The induced rules are evaluated against a predefined hypothesis and their generalizability across different system subsets is investigated. The research highlights the capabilities and limitations of ML-based rule learning in the area of software architecture, aiming to inspire further innovation in data-driven rule discovery for more intelligent software architecture practices.

Keywords: software architecture; Inductive Rule Learning; constraint learning; first-order logic; Large Language Models

1. Introduction

Beyond the inductive learning methods discussed in the original paper [1], this journal extension addresses Large Language Models (LLMs) as another area for learning formal architectural rules. The ability to effectively define and enforce software architecture rules is crucial for the development of robust and maintainable software systems. Traditional approaches to rule definition often rely on manual specification, which can be time-consuming, error-prone, and limited in their ability to capture complex architectural constraints. In recent years, there has been growing interest in leveraging Machine Learning (ML) techniques to automatically learn and detect software architecture patterns [2] from data. A notable advantage of learning software architecture rules over manual rule specifications is the elimination of the need to start from scratch. We believe that evaluating and utilizing learned rules is more straightforward than formulating rules from the ground up. Moreover, an example-based learning approach facilitates the introduction of new concepts that may not be explicitly known, for instance, through literature or expert knowledge. These concepts become explicitly understood through the learning process, which is guided by a carefully chosen set of examples.

This paper explores the application of (i) Inductive Rule Learning (IRL) and (ii) LLMs, both subfields of ML, to infer simple software architecture rules from a Knowledge Base (KB) retrieved from a systems implementation, with a ground-truth annotation of architectural rule instances.

The goal of this paper is to investigate the feasibility and effectiveness of using ML techniques to extract meaningful and actionable architectural rules from a real-world software system. By leveraging the abundance of available architectural data, we aim to
empower software developers and architects with an automated approach to rule discovery that complements traditional manual rule definition.

Our research methodology involves collecting a dataset of software architecture instances, introducing a meta-model building the foundation of collected facts from the implementation, and the preparation of facts for the examples and counterexamples based on the provided software architecture documentation.

We employ state-of-the-art IRL tools and LLMs to perform rule induction from the dataset. In addition, we formulate a hypothesis about the architectural rule we expect to learn.

The tool-induced rules are assessed for alignment with the proposed hypothesis.

In addition, we investigate the ability to infer architecture rules in different subsets of the system to indicate the possibility of applying induced rules in different parts of the same system.

The contributions of this paper lie in the exploration of inductive learning techniques and LLMs abilities specifically tailored for software architecture rule inference. By demonstrating the potential of these techniques, we hope to stimulate further research and innovation in the field of data-driven rule discovery and pave the way for more intelligent and adaptive software architecture practices.

This paper aims to address the following Research Questions (RQs):

RQ1: How effective is the application of ML techniques in inferring software architecture rules from real-world systems?

RQ2: Can the induced rules be applied in different subsets of the same system, indicating their generalizability across architectural contexts?

Summary of results: We tested different IRL tools to determine whether they could learn a predefined hypothesis. Popper, one tool, demonstrated the ability to learn a similar hypothesis. In a subsequent trial, we exposed Popper to various subsets of software architecture examples. This revealed that the chosen examples influenced the learned rule. In 21 of 26 tasks, the original hypothesis was learned, highlighting its generalizability. Deviations from the original hypothesis stemmed from subsets with simpler explanations for differentiating positive and negative cases.

LLMs faced challenges in learning the anticipated software architecture rule in line with the hypothesis. An exception was Google Bard, which successfully learned a rule closely resembling the hypothesis when being exposed solely to positive examples. Contrary to Popper’s approach, negative examples appeared to complicate the learning process for LLMs rather than offering further clarity on the task. This was evident as even Google Bard failed to formulate a rule that aligned closely with the initially stated hypothesis. In the preliminary experiment, we found that LLMs can induce a simple rule when the predicates in the examples have semantic meaning. However, if the predicates are obfuscated, the LLMs are unable to solve the task.

The outcomes of this research highlight the competencies and boundaries of IRL and LLMs in relation to software architecture. We not only present the findings but also delve into the dimensions that require increased complexity to make a substantial impact on data-driven software architecture rule inference.

The structure of this paper is organized as follows. Section 2 provides motivation and an overview of an integrated approach combining development practices with data-driven learning of architectural rules. Section 3 explores the foundations and related work in IRL, LLMs, and software architecture. Section 4 describes the experiments. Section 5 presents experiment results. Section 6 discusses the implications of the findings and outlines dimensions for increasing complexity in realistic scenarios. Finally, in Section 7, conclusions are drawn, contributions are summarized, and potential directions for future research are outlined.
2. Motivation

Software architecture is crucial in the development and maintenance of software systems, guiding their organization and implementation. However, defining and enforcing effective architectural rules becomes challenging as systems grow in size and complexity.

Traditional rule definition approaches rely on manual specification, which is time-consuming, error-prone, and limited in capturing complex constraints [3]. To address this, automated techniques are needed to discover and enforce architectural rules.

ML and data-driven techniques, in particular, show promise. They generalize from observed examples to detect recurring patterns, dependencies, and constraints.

This research aims to explore the feasibility and effectiveness of applying ML algorithms to infer software architecture rules. Our goal is to advance data-driven software architecture practices and enable more intelligent and adaptive approaches to software system design and evolution.

Figure 1 illustrates the integration of an architectural rule learning mechanism to enhance the verification of architectural consistency within development teams. On the right-hand side, we have a developer who is responsible for building and maintaining the system. In addition to writing code, the developer annotates the implemented architectural concepts, providing a foundation for performing consistency checks between the implementation and the designated software architecture for the corresponding part of the system [4,5]. This combination of manual annotation and automated rule learning ensures that architectural guidelines are adhered to, promoting a more robust and maintainable software architecture.

The process involves annotating architectural examples, which are then fed into a KB. Leveraging these examples, a data-driven learning approach [6] utilizing ML can be employed to learn rules. Architects are able to interact with the KB, configure learning tasks and tools, and validate the learned rules. The generated rules can be seamlessly integrated back into the development cycle, allowing for their practical application and enforcement within the software development process. By incorporating the learned rules, architects can ensure adherence to architectural guidelines and promote consistency and quality throughout the software system.

3. Fundamentals and Related Work

In this section, we review related work across three areas: IRL, LLMs, and software architecture. This overview sets the context for our study and illustrates the interplay between these key topics.

3.1. Inductive Rule Learning

IRL is a process that aims to learn logical rules to generalize training examples, with the objective of maximizing positive examples and minimizing counterexamples [7]. IRL
systems employ different strategies such as bottom–up, top–down, or a combination thereof to explore the hypothesis space and derive generalized rules from the training data. Learned solutions bias towards learning simple (short) rules [8]. The task of learning horn clauses is shown to be \( \Sigma^P_2 \) complete [9]. A more recent approach in IRL is meta-level IRL, where the learning problem is formulated as a meta-level logic problem that involves reasoning about programs [7,10,11] or incorporating neural networks into inductive learning [12,13], directly allowing learning tasks from raw data without the need for a pre-existing KBs.

Another recent direction involves scaling inductive learning on large KBs [14] to predict the missing links within them. This area has seen significant contributions in terms of the scalability of learning algorithms, which aim to learn hypotheses that do not necessarily apply to the entire provided KB. Instead, they focus on predicting missing links to enhance the information present in the database.

### 3.2. Large Language Models

**LLMs** are advanced types of language models known for their excellent performance in understanding and processing human language, as Fan [15] noted in 2023. These models are particularly good at making sense of information with only a few examples, especially when working with text and data tables, a capability Chen [16] pointed out in 2022. Interestingly, the larger these models become, the better they perform [17]. LLMs are also used in translating languages, where they analyze patterns in language data to produce translations, a method Brants [18] discussed in 2007. **Generative Pre-trained Transformers (GPTs)** are strong models for processing natural language. They are based on a special structure known as the transformer architecture and have many potential uses (e.g., in education, healthcare, and the entertainment industry) and challenges (e.g., high computational requirements in pre-training and inference, lack of interpretability, and ethical considerations) [19]. There has been a lot of research on GPT, especially in computer science. It is closely linked to fields like artificial intelligence, language processing, and deep learning, as Cano [20] pointed out in 2023. GPT is not just for language tasks; it is also being used in engineering design. It can help come up with new design ideas, something Gill [21] found in 2023. Zhu [22] in 2021 also noted that GPT is helpful in the early stages of coming up with design concepts.

### 3.3. Software Architecture

**Architectural design** is a critical step in software development, with a significant impact on the resulting system [23]. It involves the creation of a high-level abstraction of system topology, functionality, and behavior, which serves as the foundation for subsequent detailed design and implementation [24]. Architectural design should predominantly be guided by established and proven design patterns and best practices [25], as well as other factors like requirements [26] and stakeholder concerns [27]. Trade-offs often need to be involved to leverage conflicting goals. A well-considered design is pivotal during the creation and evolution of a system, especially when navigating changes and adaptations.

**Software architecture documentation** plays a foundational role by explicitly documenting and effectively communicating the architecture. It serves as a vital aid for decision making throughout the software system’s development and maintenance. Both informal and formal [28–34] approaches, contribute to this documentation process.

In **reverse engineering software architecture** [35,36], the objective is to glean insights into design choices, employed patterns, and inherent constraints. This endeavor not only facilitates a deeper understanding of the software system at hand but also supports its ongoing maintenance. The insights garnered from reverse engineering empower informed architectural decisions.

To our knowledge, no approach has specifically been proposed for **learning architectural rules** through examples. Existing approaches focus on architectural reconstruction, classification, and abstract representation learning from code, as well as mining software architectural knowledge from sources like Stack Overflow. For example, there are approaches
utilizing data mining and ML techniques [2,37] and others that extract knowledge from alternative sources [38]. However, there is a gap in the literature regarding directly learning architectural rules from examples.

4. Experiment

In this section, we present our hypothesis and experiments leveraging various tools to learn from real-world data in TEAMMATES [39]. We assess the effectiveness (RQ1) and transferability (RQ2) of the learned rules across different architectural contexts.

4.1. Experimental Setup

We aim to demonstrate the learning of software architecture rules using an inductive approach with a community-sourced dataset. The software system TEAMMATES provides relevant documentation [40] on a software architecture rule called isAllowedToUse [41]. It specifies that particular modules can use another modules within the system. The documentation also includes the explicit counterexamples of the rule, identifying modules that are not permitted to use certain others. The architecture documentation provides the mappings of modules to packages.

To conduct our experiment, we extracted examples from the documentation and transformed them into logic fact in the format of isAllowedToUse(usingPackage, usedPackage) and not(isAllowedToUse(usingPackage, usedPackage)).

The meta-model shown in Figure 2 is the foundation for the facts we collect from any given implementation. This describes the known domain concepts (Class and Package) among the relations between the concepts. We utilize the meta-model to derive unary and binary predicates. Unary predicates represent the concepts Class and Package, whereas binary predicates denote the relationships between these concepts. For instance, importsClass connects the concept Class to itself, indicating that Classes can import other Classes. Additionally, the cardinalities specify the nature of these relationships. In this scenario, a given Class can be imported by 0..* (i.e., zero or more) Classes. This means that, while it is not mandatory for any other Class to import a given Class, multiple Classes can indeed import it. Similarly, the cardinality at the other end of the same connection is also 0..*, implying that a class can import multiple Classes.

![Figure 2. Domain model of the types and their relationships](image)

The meta-model is intentionally small, focusing on the concepts and relations we need to define a hypothesis that covers the architecture rule isAllowedToUse(A, B). In Section 6.2, we will address the different dimensions of how the complexity of the given approach can be increased so that the approach can scale to learning rules for more complex architectural concepts.

To illustrate the relationship between the meta-model and the KB, we provide a small example using a code snippet (Listing 1) and the resulting facts (Listing 2). This example showcases how the meta-model establishes the foundation for the predicates used in the KB, as well as how concrete facts from a Java implementation are represented.

Listing 1 presents a Java code snippet, focusing on the class Gatekeeper located within the teammates.logic.api package. Notably, the implementation details of the class are excluded from the fact base as they are not relevant in this context. Additionally, the snippet highlights the import statement for the teammates.common.util.Const class, providing further insights into the class dependencies and relationships.
Listing 2 displays the resulting facts obtained from the code snippet. In total, four facts are extracted from the code. Firstly, we observe the fact representing the package `teammates.logic.api`. Secondly, there is a fact indicating that `teammates_logic_api_GateKeeper` is a class. The third fact signifies that the `GateKeeper` class imports the `Const` class from the `teammates_common_utils` package. The last fact states that `GateKeeper` is located within the `teammates_logic_api` package.

**Listing 1.** Code snippet of the Java class GateKeeper.

```
package teammates.logic.api;
import teammates.common.util.Const;
public class GateKeeper {
    // out of scope
}
```

**Listing 2.** Resulting facts in the knowledge base.

1 package (teammates_logic_api).
2 class (teammates_logic_api_GateKeeper).
3 importsClass (teammates_logic_api_GateKeeper,
teammates_common_util_Const).
4 containsClass (teammate_logic_api,
teammates_logic_api_GateKeeper).

The KB comprises 8340 facts, including 41 `package` facts, 40 `containsPackage` facts, 767 `class` facts, 767 `containsClass` facts, and 6725 `importsClass` facts.

Our **working hypothesis** is that the permitted usage of packages can be inferred by examining the import mechanism, as Java classes need to import other classes in order to utilize them. Equation (1) presents the hypothesis based on the meta-model (see Figure 2). A hypothesis consists of two parts: the rule head and the rule body.

The rule head, in our case `isAllowedToUse(a,b)`, is a single predicate, contrasting with the rule body that often consists of multiple predicates connected through logical operators. The rule body states that both `a` (the using package) and `b` (the used package) must be of the type `package`. The rules introduce two free variables (c and d) that need to be of the type `class`. Additionally, the rule body imposes constraints on the containment of variables c and d within packages a and b (using the predicates `containsClass(a, c)` and `containsClass(b, d)`), and requires the `importsClass(c,d)` relationship between the free variables to hold. In such a hypothesis, the rule head is implied by the rule body, so if the rule body holds, the rule head also holds.

\[
\forall a \forall b : isAllowedToUse(a,b) \iff package(a) \land package(b) \land \exists c : class(c) \land \exists d : class(d) \land containsClass(a,c) \land containsClass(b,d) \land importsClass(c,d)
\]

(1)

We employed several tools from the research community to learn rules for the `isAllowedToUse` architecture rule. Tools from the IRL community we use including Aleph [42] version 6, AMIE3 [43,44], AnyBURL [14] version 23-1, ILASP [10] version 4, and Popper [11] version 2.1.0. While Aleph, ILASP, and Popper were compatible with our existing KB of facts (refer to Listings 1 and 2), and adjustments were necessary for AnyBURL and AMIE3 as they are designed to learn rules on a knowledge graph.
To adapt the facts, we transformed them into a knowledge graph format comprising triples that consist of a subject, predicate, and object. For predicates with a parity of two in our KB, we performed the following transformation: \( \text{containsClass}(\text{package}, \text{class}) \) resulted in the triple \( \text{package containsClass class} \). On the other hand, unary predicates were handled differently by introducing the predicate \( \text{isA} \) and using the original predicate as the object of the triple. For instance, \( \text{package}(P) \) became the triple \( P \text{ isA } \text{Package} \).

We also utilize LLMs for the task of learning software architecture rules. The tested LLMs are Google Bard [45] version 2023.10.23, OpenAI ChatGPT 3.5 Turbo (GPT-3.5) as well as ChatGPT 4 (GPT-4) [46]. All experiments involving LLMs were conducted using the official websites that OpenAI and Google provide. The experiments did not set specific parameters such as temperature or frequency_penalty.

4.2. Conducted Experiments

Preliminary experiment with LLMs: In our initial investigation, we explored the capability of LLMs to derive first-order logical rules from given examples (and counterexamples) alongside a KB of facts. Our chosen dataset for this first study was the \textit{kinship-pi} dataset [11], focusing on grandparent relationships among individuals. This dataset utilizes the foundational predicates \( \text{father}/2 \) and \( \text{mother}/2 \), which signify the parental relationship, for instance, \( \text{father}(\text{bob}, \text{carl}) \) indicates that Bob is Carl’s father.

Our experiments were designed to evaluate the proficiency of LLMs in learning first-order logical rules from a combination of examples and provided facts. We adopted varied experimental setups. Initially, we supplied the LLMs with the complete dataset, observing their capability to assimilate the task. Subsequently, we employed an alternative interaction method, the \textit{Flipped Interaction Pattern} [47] wherein we do not provide all facts upfront. Instead, the task and examples were presented, allowing the LLMs to request specific facts, which were then supplied by a human operator.

Furthermore, we repeated these experiments with an obfuscated version of the \textit{kinship-pi} dataset. Here, predicate names and individual names were replaced with alphabet letters (e.g., \( \text{mother} \) became \( a \), \( \text{father} \) became \( b \), and so forth), effectively stripping the data of their semantic clarity.

Our preliminary findings revealed that GPT-3.5 successfully learned the rule from the original dataset but struggled with the obfuscated data. Furthermore, Google Bard had more difficulty in deriving a rule for the grandparent relationship across all preliminary experiments examples, and similarly to GPT-3.5, did not perform well with the obfuscated data.

These results suggest that LLMs may rely on semantic information, which is not explicitly provided in the task or examples, to effectively resolve the given task. This reliance seems more pronounced when dealing with data that retain their semantic integrity, as evidenced by the better performance with meaningful predicate names. Still, the general capability to induce first-order logic rules in this experiment motivated the investigation of the architectural rule \( \text{isAllowedToUse}(A, B) \) and to incorporate LLMs into the conducted experiments.

**Experiment 1:** In the first experiment, we limited the examples used to learn a hypothesis to come exclusively from the \textit{logic} package specified in the architecture. The examples we selected were constrained to directly specify the \( \text{isAllowedToUse} \) rule between packages. However, the ground data also included examples that specified the rule between packages that did not directly contain classes, but rather contained packages with classes inside them (e.g., \( \text{isAllowedToUse}(\text{teammates_ui}, \text{teammates_common}) \)). The \textbf{objective} of this experiment is to learn a simple architectural rule, denoted by Equation (1), utilizing different IRL tools and LLMs available and to answer RQ1.

Figure 3 presents an overview of Experiment 1. In its lower-right corner, we provide a key to the notation used. A box with a wavy bottom line symbolizes a document, such as our developed meta-model, software architecture documentation, among others. Activities like
the definition of a hypothesis, learning tasks, the creation of the KB, etc., are represented by boxes with rounded corners. These activities and documents are interconnected by arrows.

Figure 3. Visualization of Experiment 1.

In the upper section of the figure, the utilization of the meta-model in defining the hypothesis is depicted. This hypothesis represents the software architecture rule anticipated to be learned by the tools. It plays a crucial role in the comparison activity, where the rules learned by the tools are assessed against it.

From the software system TEAMMATES, we extract both the source code and the existing software architecture documentation. The source code, alongside the meta-model, is employed to extract logical facts that form a model consistent with the meta-model. The software architecture documentation is used to delineate both the positive and negative instances of the software architecture rule. These instances, along with the KB, are integral to the learning phase. During this phase, we format the information to meet the specific requirements of each tool being tested, in accordance with their learning specifications. Similarly to the hypothesis formulation process, the rules learned by a tool are subsequently fed into the comparison step, enabling a thorough evaluation against the original hypothesis.

We intentionally confined this first experiment to examples dealing with packages that directly contain classes. All the examples used in this experiment were packages located within the teammates.logic package. In this setup, we tested multiple tools. Depending on the capabilities of each tool, we learned rules using only positive examples of the isAllowedToUse rule, and in the cases where the tools allowed counterexamples, we also learned rules in a second run by including the same amount of counterexamples to create a balanced dataset. All tools learned on the same examples (Listing 3).

Listing 3. Examples and counterexamples used in Experiment 1.

```lisp
# positive examples
isAllowedToUse ( teammates_logic_api , teammates_logic_core ).
isAllowedToUse ( teammates_logic_backdoor , teammates_logic_api ).

# negative examples
isAllowedToUse ( teammates_logic_core , teammates_logic_backdoor ).
isAllowedToUse ( teammates_logic_api , teammates_logic_backdoor ).
```

The experimental setups for IRL tools and LLMs differ, and we elaborate on the specifics of the conducted experiments in the following section. Details regarding IRL tool configurations and interactions with the LLMs can be accessed in the accompanying reproduction package (https://github.com/schindlerc/ML-Based-Software-Architecture-Rule-Learning (accessed on 18 December 2023)).

For the IRL tools, we adhered to the provided documentation to configure them and subsequently allowed them to operate, inducing rules based on the given parameters.
In the context of utilizing LLMs for rule learning, our approach commenced with a preliminary feasibility test using a simplified dataset. This initial step served as a foundation before advancing to the more complex task of learning software architecture rules.

OpenAI offers a web-based tokenizer tool (https://platform.openai.com/tokenizer, accessed on 17 February 2024) that facilitates an understanding of how a given text can be tokenized. This tool provides information on the number of tokens and utilizes color to visually separate individual tokens within the given string. For our latest fact in Listing 2 (containsClass(teammate_logic_api, teammates_logic_api_GateKeeper)), the tool indicates a token count of 15. How the fact is tokenized can be seen in Figure 4.

![Figure 4. OpenAI tokenizer tokenizing a fact (upper part) into the tokens (lower part).](image)

Considering the scale of the KB containing 8340 facts, the total token count expands to approximately 125,100 tokens. This surpasses the token limit of GPT-3.5, capped at 16,385 tokens, and nearly reaches the maximum context window size of GPT-4 (version 1106) with a limit of 128,000 tokens. It is important to note that GPT-4 with that context window size is still in its preview stage. In [48], it is shown that current LLMs do robustly make use of information in long input contexts, focusing on the information provided in the beginning or the end of the input context, while accord lower priority to the information located in the middle of a long input contexts provided.

To overcome the challenges of the limited token window size of the models, we use prompt engineering patterns, like the flipped interaction pattern [47]. This pattern helps the LLMs by guiding them to ask questions and gather the needed information for a task. In this flipped interaction approach, instead of providing the model with all the available information, we task the LLMs with asking questions to collect needed information to fulfill the task of inducing a first-order logical rule.

In Listing 4, we show the prompt that started our conversations with the LLMs. The prompt sets the task for LLMs to ask for facts until they determine a first-order logical rule. After that, the LLMs list all available details and give examples, both positive and negative, showcasing their ability to ask questions and obtain task-related information on their own.

In the experiment, we will not allow the LLMs to ask questions forever (even if stated in the prompt) but will terminate the conversations if (i) the LLM will repeatedly ask for the same facts; (ii) it repeatedly states rules that are not correct and does not improve even when told that it is wrong.

By conducting this experiment, we sought to evaluate the capabilities of different tools and determine whether they can accurately learn the desired rule using positive examples only (Table 1) or by incorporating both positive and negative examples (Table 2) to create a balanced dataset. Learning duration and accuracy of the learned rules are given in Table 3.
Listing 4. Flipped interaction prompt.

I would like you to ask me for specific facts, so you are able to induce a first order logical rule that explains all the positive and none of the negative examples. You should ask questions until this condition is met or to achieve this goal (alternatively, forever)
Allowed predicates to ask for are:
    package/1
    containsPackage/2
    class/1
    containsClass/2
    importsClass/2

Ask me questions one at a time.
The positive examples are:
    isAllowedToUse(teammates_logic_api, teammates_logic_core).
    isAllowedToUse(teammates_logic_backdoor, teammates_logic_api).

The negative examples are:
    isAllowedToUse(teammates_logic_core, teammates_logic_backdoor).
    isAllowedToUse(teammates_logic_api, teammates_logic_backdoor).

Experiment 2: In our second experiment, we expanded the scope of the learning task by considering additional modules. In addition to the logic package, we included the storage package, resulting in a total of six packages being considered. Consequently, the number of examples increased from 6 to 30 as we examined pairwise observations regarding the permissibility of one package by another.

The objective of this experiment was to investigate the possibility of learning the same hypothesis from different subsets of examples. We aimed to determine whether a rule learned in one part of the system could also hold true for instances of the architectural rule in a different part of the system. By exploring this aspect, we sought to assess the transferability and generalizability of the learned rules across various parts within a software system to address RQ2.

Figure 5 provides a detailed overview of Experiment 2. Mirroring the methodology of Experiment 1, we constructed the KB and examples using the source code and software architecture documentation, aligned with the meta-model. However, a key distinction in Experiment 2 is the preliminary step of selecting examples, which was performed multiple times to generate various sets of examples. Each set served as input for a separate learning run using the IRL tool, Popper. During the comparison phase of this experiment, we analyzed the rules learned from the different subsets. This included a qualitative examination of the examples from which the rules were derived, offering insights into the learning outcomes.

To conduct the experiment, we randomly generated the subsets of six examples of the architectural rules and utilized Popper to learn the rules based on the given examples. To ensure balanced subsets, each randomly generated subset contained at least two positive and two negative examples.

In total, we created 26 subsets for learning rules. Each subset was used to learn a hypothesis, and the resulting hypotheses, along with the number of subsets that yielded them, are presented in Table 4.
5. Results

In Experiment 1, the rules learned are compiled into two categories and presented in Table 1 for those exclusively derived from positive examples, and in Table 2 for those formulated from both examples and counterexamples.

### Table 1. Learned rule bodies for the rule head isAllowedToUse(A, B) in Experiment 1, exclusively derived from positive examples for each tool.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Learned Rule Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aleph</td>
<td>package(B).</td>
</tr>
<tr>
<td>AMIE3 1. Rule</td>
<td>containsPackage(G, A) &amp; containsPackage(G, B).</td>
</tr>
<tr>
<td>AMIE3 2. Rule</td>
<td>isA(A, H) &amp; isA(B, H).</td>
</tr>
<tr>
<td>AnyBURL</td>
<td>isA(A, X) &amp; isA(Y, X) &amp; containsPackage(Y, B)</td>
</tr>
<tr>
<td>ILASPA (V4)</td>
<td>package(B) \lor containsPackage(C, A).</td>
</tr>
<tr>
<td>Popper</td>
<td>importsClass(D, C) &amp; containsClass(A, D) &amp; containsClass(B, C) &amp; importsClass(D, C).</td>
</tr>
<tr>
<td>Google Bard</td>
<td>containsPackage(A, C) &amp; containsClass(C, B)</td>
</tr>
<tr>
<td>GPT-3.5</td>
<td>containsPackage(Z, A) &amp; containsPackage(Z, B)</td>
</tr>
<tr>
<td>GPT-4</td>
<td>package(A) &amp; containsPackage(A, C) &amp; containsPackage(B, C) &amp; not(A = B).</td>
</tr>
</tbody>
</table>

### Table 2. Learned rule bodies for the rule head isAllowedToUse(A, B) in Experiment 1, derived from positive examples and counterexamples for each tool.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Learned Rule Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aleph 1. Rule</td>
<td>isAllowedToUse(teammates_logic_api,teammates_logic_core).</td>
</tr>
<tr>
<td>Aleph 2. Rule</td>
<td>isAllowedToUse(teammates_logic_backdoor,teammates_logic_api).</td>
</tr>
<tr>
<td>AMIE3</td>
<td>No run with explicit counter examples</td>
</tr>
<tr>
<td>AnyBURL</td>
<td>No run with explicit counter examples</td>
</tr>
<tr>
<td>ILASPA (V4)</td>
<td>UNSATISFIABLE</td>
</tr>
<tr>
<td>Popper</td>
<td>importsClass(D, C) &amp; containsClass(A, D) &amp; containsClass(B, C).</td>
</tr>
<tr>
<td>Google Bard</td>
<td>package(A) &amp; containsPackage(A, C) &amp; class(B) &amp; containsClass(C, B) &amp; not(A = B).</td>
</tr>
<tr>
<td>GPT-3.5</td>
<td>containsClass(A, B) &amp; package(A) = teammates_logic_core.</td>
</tr>
<tr>
<td>GPT-4</td>
<td>package(A) &amp; package(B) &amp; containsClass(A, X) &amp; containsClass(B, Y) &amp; not(importsClass(A, B)) &amp; not(importsClass(B, A)).</td>
</tr>
</tbody>
</table>

Furthermore, it is important to highlight that learning with counterexamples was only possible with three of the IRL tools. Among these three, only two tools successfully generated a rule. Specifically, Popper learned a rule with variables, while Aleph returned a
hypothesis consisting of the two positive examples provided but did not learn a rule with variables in the second setting. This variation in performance suggests that the capabilities of the IRL tools differed in their ability to handle the inclusion of counterexamples.

The results of Experiment 2 are presented in Table 4, summarizing learned architecture rules for the different subsets of examples.

5.1. Results of Experiment 1

Aleph includes a feature called induce_modes, which enables the induction of modes for both head and body predicates. In our study, we utilized this feature to extract the suggested modes from Aleph for the given data. Upon applying Aleph to the dataset, it learned a simple rule for \(\text{isAllowedToUse}(A, B)\) where the implication was that \(B\) represents a package (\(\text{package}(B)\)). This rule aligns with the meta-model’s restriction (refer to Figure 2), as the \(\text{isAllowedToUse}\) relation connects packages. Note that Aleph achieved accurate results solely based on positive examples.

While Aleph has the capability to learn from both positive and negative examples, the rules it learned in this scenario are not usable. The outcome generated by Aleph for positive and negative examples solely provides the positive examples, rather than learning a rule that implies \(\text{isAllowedToUse}(A, B)\) with variables \(A, B\), and potential unbound variables. As a result, the rules learned by Aleph with negative examples did not meet the desired criteria for applicability.

AMIE3 learns rules based on a knowledge graph without the explicit provision of negative examples, resulting in rules learned solely from positive examples. It is important to note that the format of the examples provided differs for AMIE3, as well as for AnyBurl, which we will discuss later.

AMIE3 learned two rules. The first rule implies \(\text{isAllowedToUse}(A, B)\) based on the existence of \(G\), which contains both \(A\) and \(B\). Given the restrictions on the selected examples, where all are packages within teammates.logic, this rule accurately reflects a valid assignment for \(G\).

The second rule learned by AMIE3 has a unique characteristic that arises from translating our meta-model into triples, as required by AMIE3. This rule states that \(\text{isAllowedToUse}(A, B)\) is implied by both \(\text{isA}(A, H)\) and \(\text{isA}(B, H)\). In the knowledge graph, there are triples such as teammates_logic_core isA package, which leads to the conclusion that the only suitable assignment for \(H\) is package. Thus, the learned rule reflects that \(A\) and \(B\) are of the same type, namely, packages.

However, these rules do not fully align with the essence of our stated hypothesis, as they do not consider the \(\text{importsClass}(X, Y)\) predicate and its connection to the head predicate, nor the involvement of the bound variables \(A\) and \(B\).

Similarly to AMIE3, AnyBURL also operates on a knowledge graph composed of triples. Consequently, the remark regarding the altered input format, where triples like teammates_logic_core isA package are used instead of \(\text{package}(\text{teammates_logic_core})\), is also applicable here. AnyBURL, unlike the other tested tools, does not require or provide a specific configuration for the head predicate of interest. Instead, it explores all possible relations within the learning phase, aiming to derive as many rules as possible. Out of the 2532 rules learned by AnyBURL, only one rule corresponds to the target head predicate \(\text{isAllowedToUse}\), which is presented in Table 1. This rule suggests that \(\text{isAllowedToUse}(A, B)\) is implied by \(A\) and \(Y\) is both an \(X\) (as indicated by \(\text{isA}\)) and \(Y\) containing a package (\(\text{containsPackage} B\)). However, similarly to the previously discussed tools, this learned rule does not capture the essence of our stated hypothesis. It is important to note that AnyBURL does not support explicit learning with counterexamples, thereby precluding a second run to refine the learned rules.

ILASP has the capability to learn rules from both examples and counterexamples. However, in our experiments, ILASP was only able to learn rules based on the two positive examples provided. When presented with the task of learning from positive and negative examples, ILASP responded with the learning task being unsatisfiable. Therefore, the rule
learned is solely based on positive examples, as shown in Table 1. The learned rule from ILASP suggests that $isAllowedToUse(A, B)$ is either implied by a disjunction $B$ being a package ($package(B)$) or by $A$ being a package contained within $C$ ($containsPackage(C, A)$). However, neither the first nor the second alternative in the rule body captures the essence of the hypothesis we formulated. Thus, the rules derived from ILASP do not align with our intended idea.

**Popper** stands out among the tested IRL tools as the only one capable of learning rules that incorporate variables in the learning task, including counterexamples, for the architecture rule $isAllowedToUse(A, B)$.

The rule learned based on positive examples alone differs from the rule learned using both examples and counterexamples. The rule derived from positive examples only implies $isAllowedToUse(A, B)$ when both $A$ and $B$ are packages. Although this rule is conditionally induced by the meta-model, it does not capture the essence of our hypothesis and is not considered suitable for abstracting the provided architectural rule examples.

The rule learned using both positive and negative examples proves to be the most promising outcome of the entire experiment. This rule includes the predicate $importsClass(D, C)$ with the free variables $D$ and $C$ in the rule body. The incorporation of this predicate and the presence of free variables enhance the rule’s potential to align with our intended hypothesis.

In addition, the rule learned by Popper connects the variables $C$ and $D$ to be classes contained in $A$ and $B$ ($containsClass(A, D)$ and $containsClass(B, C)$, respectively). This matches the last three predicates of the hypothesis in Equation (1), except for the interchange of the free variables $C$ and $D$ to $c$ and $d$, which does not alter the way that the predicates are connected in our hypothesis and the rule learned by Popper. Furthermore, it is important to note that by the interchange of $C$ and $D$ with $d$ and $c$, we are referring to the order of the variables and not the alternating use of upper and lower cases. This change in variable names does not affect the way that the predicates are connected in both our hypothesis and the rule learned by Popper.

Compared to our hypothesis, the learned rule does not contain the first part stating that the bound variables $A$ and $B$ are packages, and it does not explicitly state that $C$ and $D$ need to exist (using the existential quantifier $\exists$). However, the rule learned by Popper still captures the essence of our hypothesis by connecting the variables $C$ and $D$ as classes contained in $A$ and $B$, respectively, as well as incorporating the $importsClass(D, C)$ predicate with the free variables $D$ and $C$ in the rule body.

Existence is achieved by finding suitable assignments for the respective variables, and these unary predicates, which set the types of the variables (whether they represent classes or packages), are pre-defined by the simple meta-model and the binary predicates they are used in. Consequently, it is implicit that these aspects of our hypothesis hold in every example (and counterexample). Therefore, they do not contribute to distinguishing the provided examples from the provided counterexamples.

**GPT-3.5** demonstrated its capacity to acquire rules through two modes of the experiment: (i) from positive examples and (ii) from additional counterexamples. However, when exclusively learning from positive examples, it generated a rule that is in conflict with our meta-model. Specifically, the induced rule erroneously posits that $C$ contains a class $B$, i.e., $containsClass(C, B)$, implying an $isAllowedToUse$ relationship between a package and a class. Contrary to this, our meta-model unambiguously stipulates that the $isAllowedToUse$ dependency exclusively operates between packages.

Furthermore, the rule learned from both provided examples and counterexamples exhibits an anomalous component. It asserts that $package(A)$ equals $teammates_logic_core$, a syntactical oddity. This departure from conventional syntax is noteworthy, as it introduces a value, $teammates_logic_core$, into a predicate intended to yield a truth value of true or false, raising questions about the interpretability of such assignments.

While GPT-3.5 effectively processed the experimental data and formulated rules suggesting the presence of $isAllowedToUse(A, B)$, it was unable in both instances to induce rules that comprehensively encompassed the positive examples provided.
Unlike its predecessor, GPT-4 is capable of learning rules through the presentation of examples alone or through the inclusion of counterexamples at the outset of its training process. In both scenarios, the rules acquired during training demonstrate the capability to encompass the given positive examples, yielding flawless accuracy when positive examples alone are provided. However, when counterexamples are introduced into the training data, GPT-4 struggles to formulate a rule that effectively excludes these counterexamples, resulting in a rule set with an accuracy of 50 percent.

The comparison of our initial hypothesis with the observed rules yields intriguing insights. Firstly, the rule formed solely from positive examples has nothing in common with our hypothesis. The learned rule only expects a shared entity, Z, encompassing both A and B (\textit{containsPackage}(Z, A) and \textit{containsPackage}(Z, B)), but lacks further commonalities. In contrast, the rules derived from both examples and counterexamples exhibit greater alignment with our original hypothesis. Both not only recognize A and B as packages but also introduce X and Y as the respective \textit{containsClass} for A and B. However, a critical deviation emerges in the remainder of the learned rule. It erroneously asserts that neither \textit{importsClass}(A, B) nor \textit{importsClass}(B, A) occur, which contradicts our understanding. This is a fundamental misinterpretation, as \textit{importsClass} is intended to function with \textit{class} types, not \textit{package} types. In addition, the learned rule does not contain the part of the hypothesis that defines the \textit{importsClass} relationship between the \textit{classes} X and Y, i.e., \textit{importsClass}(X, Y).

In our first experiment, Google Bard learned rules for both setups and obtained different results. When given only positive examples, Google Bard figured out a rule that is quite similar to what we expected. It learned that A should have a \textit{class} D, i.e., \textit{containsClass}(A, D), B should have a \textit{class} C, i.e., \textit{containsClass}(B, C), and D needs to be imported into C, i.e., \textit{importsClass}(D, C). Google Bard’s rule also added that A should be a package, which aligns with our hypothesis and meta-model (see Figure 2, Equation (1)). However, the rule that Google Bard learned with counterexamples was not as good as the first one. It wrongly included the idea that B should be a \textit{class}, i.e., \textit{class}(B), which does not match our meta-model. According to our meta-model, the architectural relationship we are looking at (\textit{isAllowedToUse}(A, B)) only exists between \textit{packages}. This mistake makes the rule wrong for all the positive examples we have.

Table 3 provides the learning times and rule accuracy.

\[
\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \quad (2)
\]

<table>
<thead>
<tr>
<th>Tool</th>
<th>With Examples</th>
<th>With Examples and Counterexamples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Learning Time (s)</td>
<td>Accuracy</td>
</tr>
<tr>
<td>Aleph</td>
<td>0.010</td>
<td>1.0</td>
</tr>
<tr>
<td>AMIE3</td>
<td>0.150</td>
<td>1.0</td>
</tr>
<tr>
<td>AnyBURL</td>
<td>&lt;1.000</td>
<td>1.0</td>
</tr>
<tr>
<td>ILASP</td>
<td>4.922</td>
<td>1.0</td>
</tr>
<tr>
<td>Popper</td>
<td>&lt;1.000</td>
<td>1.0</td>
</tr>
</tbody>
</table>

# User Messages | Accuracy | # User Messages | Accuracy |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Bard</td>
<td>6 messages</td>
<td>1.0</td>
<td>44 messages</td>
</tr>
<tr>
<td>GPT-3.5</td>
<td>19 messages</td>
<td>0.0</td>
<td>54 messages</td>
</tr>
<tr>
<td>GPT-4</td>
<td>22 messages</td>
<td>1.0</td>
<td>44 messages</td>
</tr>
</tbody>
</table>

Except for ILASP, all IRL tools had learning times below one second in Experiment 1. Aleph was the fastest with 0.01 s, while ILASP took 4.922 s. In the second run, Aleph and Popper had low learning times, but ILASP took 1918 s. This long learning time can be
attributed to the exploration of the entire hypothesis space. This results in its inability to learn a rule and declaring it as unsatisfiable. All learned rules learned by IRL tools—except for Aleph where no parameterized rules were learned—achieved 100% accuracy.

For the tested LLMs, the outcomes exhibit greater variability. GPT-3.5 struggled to consistently formulate precise rules. Google Bard, in contrast, succeeded in developing an accurate rule but only for positive examples. GPT-4 demonstrated a mixed performance, achieving perfect accuracy (1.0) when working with positive examples, but its accuracy dropped to 0.5 with the inclusion of counterexamples, due to the misclassification of all counterexamples.

Assessing the learning time in interactions with chatbot-like interfaces for LLMs presents a challenge, as it is not directly comparable to the learning time of IRL tools. Therefore, we have opted to report the number of messages that the user sent to the chatbot. The total number of messages (# User Messages in the table) exchanged in these chats is approximately double this figure, considering the back-and-forth dialogue between the user and the LLM during the rule-learning process. The duration of each LLM experiment, which includes retrieving relevant responses from the LLM, composing replies, and documenting the chat history for the reproduction dataset, consistently ranged from 10 to 15 min.

After conducting experiments with tools from the distinct ML domains of IRL and LLMs, it is enlightening to highlight some fundamental differences in the learning process’s dynamics and user experience.

With IRL tools, the learning process predominantly involves setting up the test data (KB and examples) for the task at hand, enabling the tools to autonomously navigate through the hypothesis space until a rule is identified. Once initiated, this task proceeds without requiring any further input from the user or expert.

In contrast, LLMs, particularly when employing the flipped interaction mode in our experiments, initiate the learning process with an initial prompt, followed by an iterative exchange of messages between the LLM and the user. This interactive approach not only accommodates a diverse range of messages from both parties but also supports adaptive responses during the rule-learning phase. We recognize significant potential in further exploring these interactive possibilities, as they have a profound impact on both the learning journey and its outcomes.

Regarding RQ1, our findings indicate that Popper, utilizing examples and counterexamples, learned a rule closely aligned with the proposed hypothesis. However, other IRL tools learned rules that did not fully capture the essence of the architectural rule, yet still encompassed the presented examples. Notably, these tools generated rules almost instantly for the small-scale experiments, achieving perfect accuracy. In the context of learning software architectural rules, LLMs exhibited significant limitations. Despite being extensively trained on substantial datasets, LLMs frequently encounter challenges in accurately learning software architectural rules, closely aligning with our hypothesis. Notably, only in one experimental setup (Google Bard, absent counterexamples) did the essence of our hypothesis become evident in the learned rule. We observed ML techniques from IRL and LLMs that showed promise in learning effective rules in our experiments. However, many tested approaches tended towards simpler explanations for the examples provided. This inclination resulted in the derivation of basic rules that ultimately fell short of meeting our expectations.

5.2. Results of Experiment 2

For the second experiment, we solely focused on using Popper, as it was the only tool from Experiment 1 that learned a rule closely aligned with our hypothesis on examples and counterexamples. The results of Experiment 2, presented in Table 4, showcase the learned rule and the number of subsets that led to its discovery. To summarize the learned rules, we applied the commutative law by reordering the body predicates and aligned the free
variables. Interestingly, we observed that 21 out of 26 subsets produced the same rules as Popper learned in the first experiment, considering both examples and counterexamples.

Table 4. Experiment 2: Popper’s learned rule bodies for the rule head \( \text{isAllowedToUse}(A, B) \) and their amount for different example subsets.

<table>
<thead>
<tr>
<th>Learned Rule Body</th>
<th># of Subsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{containsClass}(A, C) \land \text{containsClass}(B, D) \land \text{importsClass}(C, D). )</td>
<td>21</td>
</tr>
<tr>
<td>( \text{containsPackage}(C, B) \land \text{containsPackage}(C, A). )</td>
<td>2</td>
</tr>
<tr>
<td>( \text{importsClass}(D, C) \land \text{package}(A) \land \text{containsClass}(B, C) \land \text{importsClass}(C, D). )</td>
<td>2</td>
</tr>
<tr>
<td>( \text{containsClass}(B, C) \land \text{importsClass}(D, E) \land \text{containsClass}(A, D) \land \text{importsClass}(D, C) \land \text{importsClass}(C, E). )</td>
<td>2</td>
</tr>
</tbody>
</table>

For the remaining five cases where different rules were learned, we examined the examples within each subset to determine the reasons behind the variation. In two instances, Popper derived the rule \( \text{isAllowedToUse}(A, B) \) from a free variable \( C \) that contained both packages \( A \) and \( B \). These instances corresponded to accurate rules, as all the examples within each subset belonged to the same package (either \text{teammates.logic} or \text{teammates.storage}). Notably, this was not the case for the counterexamples, which contributed to the differentiation in learned rules.

In the case of the other two rules that have been learned (twice and once, respectively), we noticed a common counterexample \( \text{not(isAllowedToBeUsed(teammates_logic_core, teammates_storage_entity)} \) that appeared in all three subsets. However, we believe that this particular negative example may be inaccurately labeled or could represent an inconsistency within the documented architecture and the implementation.

We identified a class \( \text{FeedBackResponseLogic} \) within the package \text{teammates.logic.core} that imports and interacts with objects from \( \text{FeedbackResponse} \), which is located in the package \text{teammates.storage.entity}. This violates the architectural rule, as the expected behavior suggests no allowance to use objects from the other package. However, in the implementation, we observed the usage of classes from both packages, which contradicts the rule of not being \text{allowedToUse}. Due to this inconsistency, the expected rule fails to distinguish between the provided examples and the counterexamples. This violation raises doubts about the accuracy of the labeling and highlights a potential discrepancy between the documented architecture and its actual implementation.

As a consequence, the rules learned in this particular case tend to be more complex. This is primarily because the examples and the inconsistent counterexample exhibit greater similarity to each other in terms of the facts that can be represented using the simple meta-model (Figure 2). Popper needs to learn a more intricate rule to differentiate between the examples and counterexamples within the specific subsets.

In response to RQ2, our findings demonstrate that Popper is capable of inducing rules from various subsets of examples within the same system, suggesting a degree of generalizability across different architectural contexts. However, it is important to note that the specific subset employed can influence the learned rules and, consequently, their generalizability. For instance, if the IRL tool can find a simpler explanation, such as “all positive examples have packages located in the same package”, it prioritizes this shorter explanation over more nuanced rules.

6. Validity Considerations and Future Directions

While our experiments have indeed produced promising results, it is essential to recognize and address potential factors that could impact their validity.
6.1. Threats to Validity

Limited Dataset Scope: The effectiveness demonstrated by our model on the small, single-source dataset might not entirely encapsulate its potential when confronted with larger and more diverse datasets. The inherent constraints of the dataset’s scale and diversity could influence the model’s performance on real-world, multifaceted data distributions.

Single Architectural Rule: Our investigation solely focused on a solitary, straightforward software architectural rule. As a consequence, the applicability of our findings to the learnability of considerably more intricate rules, characterized by greater complexity and a wider spectrum of concrete implementations, may be limited based on the scope of the conducted experiment.

With respect to LLMs, solely one interaction tactic (the Flipped Interaction Pattern) was used in the experiments, as the token scope is a limiting factor on how to provide software architecture-related tasks on learning software architecture rules.

By acknowledging and addressing these aspects, we ensure a comprehensive understanding of the boundaries and implications of our study’s outcomes. This discernment is vital for translating our findings into broader contexts and for guiding future research in the field.

In the following section, we outline ways to enhance the complexity of our approach and tackle the stated threads.

6.2. Dimensions of Complexity Enhancement

To elevate the demonstrated approach of learning software architecture rules to a realistic level, there are several dimensions in which complexity needs to be increased.

One aspect is increasing the size of the meta-model. The current meta-model introduced in the experiment has limitations in terms of the types and relations it represents. It lacks the ability to encompass the fundamental concepts of object-oriented programming, such as inheritance, attributes, methods, and their visibility. Consequently, a significant amount of information from the implementation is currently not captured in the KB. To address this, a valuable extension of the meta-model is required to incorporate necessary information, enabling the expression of more complex architectural rules. By expanding the meta-model to include these crucial aspects of object-oriented programming, we can enhance the richness and depth of the KB. This, in turn, will allow for the learning of more sophisticated architectural rules that better align with the intricacies of real-world software systems.

Another dimension in increasing the complexity of learning architectural rules involves the ability to utilize quantifiers in the learned rules. The inclusion of quantifiers enables the learning of more precise rules by allowing the expression of conditions that must hold for every occurrence and we gain the capability to capture and enforce more nuanced and specific patterns within the software architecture. Ultimately, the use of quantifiers enhances the precision and accuracy of the learned rules, enabling a deeper understanding of the architectural properties of the system.

Addressing the need to handle more heterogeneous examples represents a significant dimension of increasing complexity. This requirement arises from various factors. One contributing factor is the expansion of the meta-model and the growing complexity of the collected facts derived from the implementation. As the KB expands in size and intricacy, we can expect the examples to exhibit a greater diversity compared to the relatively constrained flexibility observed in the conducted experiments.

Another reason for the necessity of handling more heterogeneous examples stems from the nature of complex architectural rules. While we have demonstrated the ability to learn simple software architecture rules using only a few examples and counterexamples, this may not hold true for more intricate rules. Therefore, addressing the challenge of handling more heterogeneous examples is crucial for advancing the learning of software architecture rules. This task becomes particularly pertinent considering the expanding meta-model and the inherent complexities associated with real-world architectural scenarios. By tackling this challenge, we can enhance the robustness and applicability of learned architectural rules in diverse software development contexts.
7. Conclusions and Future Work

In conclusion, this research has shed light on the potential of inductive learning techniques for inferring software architecture rules from known examples. Our findings highlight the capabilities of IRL and LLMs in capturing and detecting a simple architectural rule. Through experiments and analyses, we have shown that these techniques can learn meaningful rules that align with the expected architectural hypotheses, albeit with some limitations and variations in learned rules solely based on (positive) examples or on specific subsets of examples. The results are promising, and in combination with the efforts of IRL approaches to learning in large KBs and the fast pace in LLMs, intriguing questions about the scalability of the proposed approach arise. Moreover, we identified dimensions for increasing the complexity of rule inference, including expanding the meta-model to encompass more architectural concepts, handling heterogeneous examples, and incorporating quantifiers for more precise rule expressions. Future work aimed at addressing the complexity dimensions discussed could involve conducting a more extensive test to explore the scalability of the approach. Additionally, this would provide an opportunity to further investigate and mitigate current threats to validity. These research outcomes provide valuable insights in the field of data-driven software architecture.

By further exploring and advancing learning approaches, we can unlock the potential of (semi-)automated rule discovery, leading to more robust, maintainable, and adaptive software systems. Looking ahead, future research can focus on addressing the identified dimensions, refining the rule learning process, and exploring novel techniques to enhance the expressiveness, scalability, and generalizability of learned architectural rules. We recognize the promising potential of hybrid methodologies combining LLMs with IRL strategies. Specifically, we aim to explore various interaction modes with LLMs, such as advanced prompting techniques. This will include more active interactions (from the user) and providing additional comprehensive information in the prompts such as the typesystem of the meta-model or the whole universe of entities of a given task. Another area for enhancement involves integrating validation checks into the LLM learning process. This will ensure that any learned rules are subject to syntactic verification, and rules that violate the constraints of the typesystem or deviate from the principles of constructing first-order logical rules will be rejected. By continuing to push the boundaries of data-driven rule inference, we can empower software developers and architects with powerful tools for effective software architecture governance and evolution.

Author Contributions: Conceptualization, data curation, formal analysis, investigation, software, methodology, validation, investigation, writing—original draft preparation, writing—review and editing, visualization, and project administration C.S.; funding acquisition and supervision, A.R. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data (knowledge base), logs, and instructions on how to use the tools are released in the GitHub-repository https://github.com/schindlerc/ML-Based-Software-Architecture-Rule-Learning (accessed on 18 December 2023).

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

- **GPT**: Generative Pre-trained Transformer
- **IRL**: Inductive Rule Learning
- **KB**: Knowledge Base
- **LLM**: Large Language Model
- **ML**: Machine Learning
- **R**: Research Question
References


Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.