LLMs Choose the Right Stack: From Patterns to Tools

Sebastian Copei*†, Oliver Hohlfeld*, Jens Kosiol‡ and Aleksandar Ristoski†
*Distributed Systems
University of Kassel, Kassel, Germany
Email: {sco, oliver.hohlfeld}@uni-kassel.de

†Innovationfield Digital Ecosystems
Fraunhofer IEE, Kassel, Germany
Email: {sebastian.copei, aleksandar.ristoski}@iee.fraunhofer.de

‡Philipps-Universität Marburg
Marburg, Germany
Email: kosiolje@mathematik.uni-marburg.de

***Abstract*—Choosing suitable architectural patterns and the technologies that implement them is a complex design task. We evaluate how well current LLMs can support such decisions by empirically evaluating six LLMs (five open-source, one closed-source) on three scenarios: (i) naïve versus prompt-engineered pattern recommendation, (ii) decision-tree-guided selection via the CAPI method, and (iii) mapping patterns to concrete tools from a supplied list. We assess reasonableness, consistency, pattern specificity, and output structure. We show that even minimal prompts yield reasonable suggestions, while prompt engineering improves focus on architectural (rather than low-level design) patterns and consistency. CAPI guidance expands coverage and approaches human-expert performance, though models exhibit a strong bias toward micro-services and tend to over-suggest patterns. All models propose plausible tools when a curated list is provided. Overall, LLMs—especially when combined with structured prompts and decision-tree guidance—can meaningfully augment architectural decision-making, while highlighting the need for tighter output control and broader, less biased pattern coverage.**

***Index Terms*—LLM, architectural patterns, decision making.**

I. Introduction

Large language models (LLMs) such as ChatGPT, Claude, and GitHub Copilot have become pervasive tools in contemporary software-engineering practice. Developers now rely on LLMs for code completion, bug fixing, or test-case generation [1], [2]. The widespread reliance on these models has even given rise to the colloquial notion of vibe coding, a meme-originated term that now captures the tendency to over-use of LLMs in software development. This rapid adoption raises an obvious next question: *Can LLMs also assist with the high-level architectural decisions that precede coding?*

Prior decision-support approaches for architecture rely on rule bases, inference engines or questionnaires (e.g. [3]–[5]) that assists in selecting architectural designs—most recently the *Comprehensive Architecture Pattern Integration (CAPI)* method [6]. Recent work has started to probe LLMs for architectural diagrams, analyzing existing designs, or reconstructing Architecture Decision Records [7]–[11]. However, no study has yet investigated whether LLMs can (1) recommend *architectural patterns* for a given set of requirements and (2) map these patterns to concrete implementation technologies—capabilities that are crucial in industrial projects [12].

We take a first step to close this gap by systematically evaluating off-the-shelf LLMs on the task of providing architectural recommendation. Using the four evaluation scenarios introduced in CAPI [6], we compare: *(i)* the pattern and tool suggestions generated by 6 state-of-the-art LLMs, *(ii)* the same suggestions when the LLMs are *primed* with the decision-tree output produced by CAPI, and *(iii)* the ground-truth recommendations provided by human software-architecture experts in the original CAPI user study. These experiments enable us to address the following research questions:

- RQ1 How well does an LLM perform in suggesting architectural patterns?
- RQ2 Can the results of an LLM be improved if it uses a decision tree to suggest architectural patterns as prompt engineering?
- RQ3 How well does an LLM derive concrete tools from architectural patterns?

The results indicate that LLMs already achieve promising levels of accuracy on architectural recommendation tasks even when only minimal prompt engineering is applied. Moreover, when the LLMs are primed with the decision tree output generated by CAPI, their precision improves further, highlighting a synergistic relationship between knowledge-based (CAPI) and data-driven (LLM) techniques for architectural decision support. Our contributions are as follows.

- We show that even with minimal prompting, off-the-shelf LLMs can generate suitable architectural pattern recommendations, as long as the user describes the target application with its requirements.
- We show that the quality of the recommended patterns can be increased through more informative prompts. In particular, by feeding the LLM the questions from the CAPI decision-tree [6]—the same questions a human architect would answer—we let the model act as the subject in the CAPI process. This indirect approach yields substantially better results than asking the LLM to

produce architectural recommendations outright, showing that LLMs can reliably replace human engineers in parts of the design stage and reduce the need for extensive human surveys.
- We show that LLMs can also suggest suitable tools that implement architectural patterns, if a list of selectable tools is given, to close the circle from designing to implementing a software system.
- We publicly release our code, prompts, and data that enables the replication of our work at [13].

II. Related Work

In this paper we investigate the extent to which LLMs can support architectural design decisions, specifically by recommending appropriate architectural design patterns and suggesting suitable implementation tools. Next, we *(i)* survey prior efforts that address similar objectives without employing LLMs, and then discuss *(ii)* recent studies that leverage LLMs for architectural reasoning and design assistance.

Regarding *(i)*, relatively recent work includes the development of a decision model that suggests a foundational architectural style based on quality attributes desired from the system [3], a study that maps patterns for microservices to problems they solve [4], and a decision support system that leverages an inference engine to derive architectural design pattern suggestions from requirements [5]. Moreover, CAPI [6] guides users through a series of questions regarding the envisioned system and outputs a list of architectural design patterns, suitable to implement that system (see Section III).

Regarding *(ii)*, Schmid et al. provide a very recent survey on the use of LLMs for software architectural tasks [14]. In the following, we focus on works that are most closely related to ours. Maranhão and Guerra have suggested a catalog of five prompt patterns and a sequence to use these in when prompting LLMs for architectural design suggestions [15]. To analyse the architectural knowledge contained in LLMs, Soliman and Keim systematically test GPT's knowledge of the Hadoop HDFS system [10]. In a similar vein, Guerra and Ernst assess GPT's architectural knowledge by using it to thoroughly analyze (following Bloom's taxonomy) a self-developed application that uses the VIPER architectural style [7]. Dhar et al. use five projects with Architecture Decision Records (ADRs) to see in how far LLMs can reproduce the ADR's decisions from the given context [9]. Rubei et al. use LLMs (Gemini and ChatGPT) to make repair suggestions when a software architecture (documented in a self-designed DSL) does not conform to a reference architecture [11]. Jahić and Sami use ChatGPT to create architectural designs for five test projects according to the C4 model (including diagrams) [8]. They evaluate the variance of ChatGPT's output across multiple runs and let the original implementers of the projects assess the suggested designs. Regarding patterns, LLMs have been evaluated for their ability to recognize [16] or recommend [17] the use of the classic GoF design patterns [18].

Whereas some authors report very successful performance of their employed LLM(s) [7], [9], [11], reported problems include superfluous suggested actions [11]; only moderate quality, with answers being wrong or overly general [10]; or unreliability because of variance in and incompleteness of answers [8]. Furthermore, most studies (with the exceptions of [9], [16]) evaluate their approach using only one or two models. Thus, further studies are needed to build up knowledge for which software design tasks which LLMs can be successfully used and in which way(s). To the best of our knowledge, no prior work has examined the use of LLMs for recommending architectural design patterns—or the corresponding implementation tools—a task of clear industrial relevance [12] that we aim to address.

III. CAPI 101

To evaluate the capability of LLMs in offering architectural recommendations for software projects, we utilize the Comprehensive Architecture Pattern Integration (CAPI) approach [6]. CAPI is designed to generate concrete architectural suggestions—particularly for web-scale, distributed systems—and its recommendations have been assessed in user studies with professional software engineers [6]. By leveraging the four evaluation scenarios defined in CAPI, we are able to assess the patterns recommended by off-the-shelf LLMs against those by software engineers in the CAPI study.

Capi 101. It assumes that an overall architectural style (e.g., monolithic, client-server, or microservices) has already been chosen. Based on a decision-tree, CAPI recommends concrete architectural patterns such as Containerization, API Gateway, or Shared Database Server. The suggested patterns stem from a systematic review of standard patterns commonly employed in microservice-based systems [19]. Figure 2 shows a fragment of the decision tree that can lead to two alternative recommendations. For example, answering 'yes' to the first two questions and 'no' to the third yields the patterns Service Discovery and Service Registry. The final questions in each branch act as contraindication checks, ensuring that a pattern is truly required for the given context.

4 Evaluation Scenarios. To compare LLM suggestions with the original CAPI results that were assessed in user studies, we reuse the same four evaluation scenarios that were employed in the CAPI user study with software engineering experts [6]. Each scenario describes a concrete system, its requirements, and a target architectural style (the first two scenarios are fixed to Monolithic and Microservices; the latter two are open to several styles). By feeding the unchanged scenario texts to the LLMs we obtain an apples-to-apples comparison for RQ1–RQ3 and can isolate the effect of providing the CAPI decision-tree output as additional prompt context (RQ2).

IV. Methodology

In this section, we outline the design of our experiments.

A. General Setup

For all experiments we employed the six LLMs listed in Table I. As Open-Source models, we utilized *Deepseek-r1*, *Meta-Llama-3.1.8b-Instruct*, *Mistral-Large-Instruct*, *Codestral-22b*,

The Flower Shop

You are working for a small florist shop in Flowerville. Your shop has multiple stores around the town. The company decided that a webshop should be implemented to better reach the people in Flowerville. There are no plans for overarching distribution beyond the city limits. Flowerville is a medium-sized town with roughly 90,000 inhabitants.

Streamland

You are working in a big software engineering team at the company Streamland. The company was founded freshly and acquired most of the streaming licenses for movies and series worldwide. They want to build a new platform where all the current streaming services are combined, so customers only need a single service to stream all movies and series. The application should be available for all devices, like TVs, smartphones, or PCs.

Company Administration

You are working in a corporate group. The CFO comes up with a new financial saving plan. Hereby the CTO decides that the present hardware resources need to be used more cost efficient. Administration services like E-Mail hosting, document management, business trip management, or vacation management should be deployed into a suitable infrastructure.

Accord

You are working in a middle-sized enterprise. As a learning from the COVID pandemic, you must develop a new chat and conference tool. The tool should be tailored for business usage. Core features should be different chat services and a video/voice channel service.

Fig. 1. Evaluation scenarios used for the experiments.

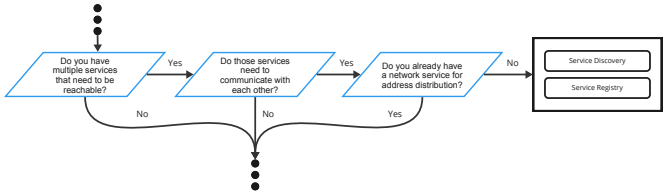

Fig. 2. Example question series from CAPI

and *Qwen2.5vl-72b-Instruct*, and as Closed-Source models *ChatGPT-o4*. Throughout the paper we refer to each model by its short abbreviation. All models were queried with the default decoding parameters, i.e., *temperature* = 0.5 and *top_p* = 0.5. We selected these models as prominent examples for often used LLMs, all offered via Chat AI [20][1]. Chat AI offers different APIs to interact with Open-Source and commercial models, such as GPT, for research. We employ qualitative evaluation methods for our analysis. For each experiment, we describe the chosen methods in the following sections.

TABLE I
LLMS USED FOR OUR EXPERIMENTS

Model	Abbr.	Open-Source	Closed-Source
ChatGPT-4o	GPT		x
Codestral-22b	Code	x	
Deepseek-r1	DS	x	
Meta-Llama-3.1.8b-Instruct	Llama	x	
Mistral-Large-Instruct	Mistral	x	
Qwen2.5vl-72b-Instruct	Qwen	x	

B. Experiment 1: General performance of LLMs to suggest architectural patterns.

The first experiment assesses how well the LLMs can generate reasonable architectural-pattern suggestions for a given scenario. We evaluate two prompt variants: the first one being a *naïve* prompt and the second being an *improved* one, using at least basic prompt engineering [21]; for the second prompt,

[1]https://kisski.gwdg.de/en/leistungen/2-02-llm-service/

> Hi, I am a student junior developer and I have some work to do. I will give you a concept and you tell me what tools, pattern designs I should use, maybe?

Fig. 3. *naïve* prompt for Expt. 1

> Hello, I am a software architect, and I must create an abstract software architecture for the following scenario. As a result, I need a list of software architecture patterns that I must consider for the scenario. First, choose the most suitable style from the architecture styles: monolithic, server/client, and microservices, and then suggest suitable software architecture patterns. By software architecture patterns, I mean, for example, patterns such as “Containerization”, “CI/CD”, or “Backend for Frontends”.

Fig. 4. *Improved* prompt for Expt. 1

we provide context, a clear role, and examples for the expected results. For both prompts, we use the system prompt “*You are a helpful assistant*”. We used Fig. 3 as the *naïve* prompt and Fig. 4 as the *improved* prompt. For both the *naïve* prompt and the *improved* prompt, we combined the prompt with each of the scenarios and sent the combination of prompt and scenario to the LLM. Each model is queried ten times per prompt variant, with a fresh chat session started for every query to avoid cross-run contamination.

To evaluate the experiment, we assessed each model on three orthogonal properties—Reasonable, Variance, and Patterns—using a three-point ordinal scale (1 = low, 3 = high).

Reasonable captures the overall usability and practicality of an answer. A score of 1 indicates a brief reply with no justification, a score of 2 reflects a fitting but incomplete recommendation, and a score of 3 denotes a near-complete analysis that extracts implicit requirements from the scenario, reasons about them, and proposes an appropriate (architectural) solution. This metric therefore measures how well the LLM understands the task and translates scenario details into sound (architectural) advice.

Variance quantifies the consistency of a model's output across the ten repetitions of the same prompt. A score of 1 means the responses differ both in structure and in the suggested patterns, a score of 2 indicates a stable structure but varying recommendations, and a score of 3 signifies nearly identical answers in both form and content. By measuring variance we gauge the indeterminism inherent to each LLM, an important factor because even with temperature = 1.0, stochasticity can prevent reproducible results [22].

Patterns evaluates whether the model's answer references the correct type of pattern. A score of 1 means the LLM answered without mentioning a pattern after all. A score of 2 means the LLM mixes design and architectural patterns, and a score of 3 means the LLM answers only with architectural patterns. With the Patterns property, we can align the overall reasonability of the LLMs with our expected outcome of architectural patterns. An LLM can give a reasonable answer to our input without mentioning an architectural pattern or even without any architectural concerns, if the LLM answers from the perspective of a developer.

C. Experiment 2: Decision-tree guided prompting

In the second experiment, we evaluate we can improve the results by letting the LLM act as a subject that is being asked questions from the CAPI decision-tree [6] that selects a software architecture. Unlike Experiment 1, where the LLM is asked to produce patterns directly, here the LLM is presented with the full sequence of CAPI questions—a lengthy questionnaire that a human architect would answer to arrive at an architecture recommendation.

Each LLM needs to answer questions with "yes" or "no". In case the model answered in another way, we repeated the question until we received a "yes" or "no" as an answer. For some questions, the LLM needs to select a single option from a list of options. Also in this situation, we repeated the question until the LLM answered with a single option. Furthermore, we do not repeat the overall experiment multiple times, because of the simplicity of the answers. To further enforce that the LLM is only answering with "yes" or "no", we set the following system prompt: *Answer each question clearly and concisely. For yes/no questions, be decisive*. As the initial question of CAPI is about the desired architectural style (Monolithic, Server/Client, Microservices), we have prepared the initial prompt pictured in Fig. 5. Here, {*scenario*} is a placeholder for the current scenario for the current tree traversal.

To evaluate this experiment, we reuse the results of the study that was conducted to refine the CAPI decision tree [6]. The study involved ten participants and measured two metrics: (1) the number of suggested patterns and (2) a distance score. We initially used this score to determine how a specific change on the underlying decision tree of CAPI will change the resulting set of suggested patterns. For each of the scenarios, we created a list of desirable patterns that must, could, or must not be part of the resulting set of suggested patterns. For each pattern in the list of suggested patterns, we give penalty points. If a pattern is required (must) and the CAPI output contains it, we assign 0 penalty points; if the required pattern is missing, we add 1 point. For patterns that are optional (could), we award 0.5 points when they appear, but we do not penalize the system for omitting them. Patterns that are are not suitable (must-not) incur a heavier penalty: we add 1.5 points when such a pattern is suggested, and 0 points when it is absent. In this scheme a higher total score indicates a larger distance between the set of patterns that we consider desirable and the set actually suggested by CAPI (or by an LLM). The asymmetry of the penalties reflects the intuition that suggesting an unsuitable pattern can increase system complexity more severely than simply omitting an optional one; therefore must-not patterns receive a larger penalty than could patterns. By computing this distance score we can quantify how closely the LLM-generated recommendations match the reference list and compare the LLMs' performance with that of the human participants in the original study. As this list of desired patterns is not objective and only represents our implementation approach, we have also included the results of the former study, the evaluation of the scores and images of the target architectures for each scenario in the research bundle to support reproducibility [13].

> Hello, you are a software architect, and you must create an abstract software architecture for the following scenario {scenario} As a result, you need to answer these questions regarding the software architecture patterns that you must consider for the scenario. You are looking for the most suitable style from the architecture styles monolithic, server/client and microservices and then answer to get a suitable software architecture patterns. By software architecture patterns I mean, for example, patterns such as "Containerization", "CI/CD" or "Backend for Frontends". 1. What is your target Architecture: 1. Monolithic 2. Server - Client 3. Microservices Please only answer with the choices provided.

Fig. 5. Initial prompt for CAPI-structured prompting

D. Experiment 3: LLMs' tool suggestions

As our last experiment, we tested the capabilities of the LLMs to suggest tools that implement the previously suggested patterns. We reused the enhanced prompt from the first experiment and extended it with the following sentence: *Also, for whatever pattern you suggest, here is the list of tools you need to pick to fit the pattern:* {*tool_list*}. While {*tool_list*} is a prepared list of all tools from the CNCF Landscape[2], structured as a JSON list. By providing such a list of tools, we want to increase the overall quality of suggested tools by the LLM. Furthermore, we added a description of the expected output to the final prompt. The LLM should structure its answer with JSON so that the answer could be processed further by a tool such as CAPI. Figure 6 shows the added prompt. We used the same system prompt as in the first experiment.

[2]https://github.com/cncf/landscape/blob/master/landscape.yml

And a very important thing – your answer HAS to be a structured json and nothing more. No additional text from your side. Output: { “architecture style”: “Monolithic / client - server / microservices” (which ever you choose out of these 3), “patterns”:[{’name’:’pattern1 you suggest to be used’,’tools’:[list of 2 tools you suggest to be used for this pattern –>pick them from the list below]}, {’name’:’pattern2 you suggest to be used’,’tools’:[list of 2 tools you suggest to be used for this pattern –>pick them from the list below]}, {’name’:’patternN you suggest to be used’,’tools’:[list of 2 tools you suggest to be used for this pattern –>pick them from the list below]},]}

Fig. 6. Prompt to enforce structured output

To evaluate the third experiment, we will first check whether the LLMs can produce a structured output in the JSON format that needs no further refinement to be used in another tool. After that, we will check the quality of the suggested tools and their reasonability.

V. Results

In this section, we present the results of our three experiments. We release the raw data of the experiment, including all outputs and conversation histories, the used Python scripts to automate the experiments and the process as a research bundle at Zenodo [13]. We will outline the interpretation of these results in Sect. VI.

A. Experiment 1: General performance of LLMs to suggest architectural patterns

As described in Sect. IV-B, we performed each prompt ten times to assess the variance in answers for each model, resulting in a total of 480 prompts for four scenarios on six models. Table II shows the overall performance of the models in comparison. Furthermore, we ranked the LLMs once for the naïve and the improved prompt. The performance of each model was consistent across all four scenarios for all metrics described in Sect. IV-B. Therefore, we do not distinguish between the scenarios in Table II.

To provide a clearer context for the results shown in Table II, we present a few representative excerpts from the LLMs’ answers to Scenario 1 using the naïve prompt. The excerpts illustrate both low- and high-scoring behaviors for the three evaluation dimensions: Reasonable, Variance, and Patterns.

Reasonable. *DS* summarizes the scenario, extracts the key requirements and offers a detailed description of the implementation processes, e.g., (“*...Customer adds products to cart. We can reserve the items for a short time...*”). In contrast, *Qwen* lists a few technologies without any explanation and merely names the required processes, e.g., (“*...you may want to consider implementing features such as user authentication, shopping cart...*”).

Variance. *Code* shows a pronounced deviation between its two runs. For the same question it first answers (“*Design Patterns: ...you might consider using a responsive design pattern*”) and later replies (“*Patterns:...using MVC..., making the application easier to maintain...*”). In comparison, *Qwen* is consistent, producing identical answers in both runs (“*Front-end: HTML, CSS, JavaScript, and...React or Angular.*”).

Patterns. The contrast between naïve and improved prompts is most evident for *Mistral*. With the naïve prompt the model returns only design-pattern names: (“*Design Patterns: MVC, Repository Pattern,..., Dependency Injection*”). With the improved prompt it produces architectural-level recommendations: (“*... Containerization,..., CI/CD,..., Service Discovery,...*”).

In the overall ranking, *DS* achieved the highest total score, while *Qwen* was the weakest overall. In terms of raw points, *Code* also scores low, but because its answers are more reasonable than those of *Qwen* we rank *Code* above *Qwen*. The models *GPT*, *Llama* and *Mistral* performed almost identically and are thus listed alphabetically.

Effect of prompt engineering. Comparing naïve and improved prompts reveals trends. The *Reasonable* score is largely unaffected by prompt refinement. The *Variance* score improves for *Code* when the prompt is engineered. Except for *Qwen*, all models tend to produce more architectural patterns (rather than generic design patterns) with the improved prompt.

Qualitative observations. DS often supplies database schemas that implement the suggested data model, whereas *Qwen* mentions patterns only briefly and provides minimal justification. Even when *Qwen* is perfectly consistent across ten runs, the variations are limited to single-word changes (e.g., “Although” vs. “While”) or occasional extra pattern suggestions. Many models, including *DS*, still favor Gang-of-Four design patterns [18] over true architectural patterns. *Code* sometimes proposes abstract concepts rather than concrete patterns. With the improved prompt, the likelihood of receiving architectural patterns increases. For example, *GPT* shifts from repeatedly naming MVC, Singleton, and Factory (design patterns) in the naïve setting to suggesting Microservices, API Gateway, and Containerization (architectural patterns) after prompt refinement.

Pattern diversity. Even with the improved prompt, the LLMs tend to output a relatively short list of patterns, typically 5 to 10. *DS* follows this trend as well, although it provides considerably richer justification and follow-up description for the architect.

B. Experiment 2: Decision-tree guided prompting

In the second experiment, every model followed the CAPI decision-tree correctly for all scenarios. Notably, the pattern lists produced by combining LLMs with CAPI were more accurate than the direct-prompt results from the first experiment. To interpret these results, we compared the LLMs’ number of suggested patterns and their distance scores with those of the human participants (Sect. IV-C). Table III reports the mean number of patterns and the mean distance score for participants who selected the same architectural style as the LLM. The number of such participants is shown in parentheses in the

TABLE II
ANSWER QUALITY OF LLMS IN EXPERIMENT 1

Model	Naïve prompting			Improved prompting			Total
	Reasonable	Variance	Patterns	Reasonable	Variance	Patterns	
DS	3	2	2	3	2	3	15
GPT	2	2	2	2	2	3	13
Llama	2	2	2	2	2	3	13
Mistral	2	2	2	2	2	3	13
Code	2	1	2	2	2	3	12
Qwen	1	3	2	1	3	2	12

"Participants" column. We only compare to participants who chose the same architectural style since, in CAPI, the resulting number of patterns highly depends on that.

TABLE III
EXPERIMENT 2: LLMS PROVIDE BETTER SUGGESTIONS WHEN FOLLOWING A CAPI DECISION TREE

Model	Sce.	Style	Patterns		Distance	
			LLM	Participants	LLM	Participants
GPT	1	S/C	14	9 (7)	8.5	4 (7)
	2	MSA	29	32 (8)	12.5	10 (8)
	3	MSA	27	19 (7)	12	8 (7)
	4	MSA	33	30 (3)	11.5	11 (3)
Code	1	MSA	29	20 (1)	8.5	7.5 (1)
	2	MSA	34	32 (8)	9	10 (8)
	3	MSA	32	19 (7)	13	8 (7)
	4	MSA	33	30 (3)	11.5	11 (3)
DS	1	Mono	7	10 (2)	7	7 (2)
	2	MSA	14	32 (8)	26	10 (8)
	3	MSA	11	19 (7)	8	8 (7)
	4	MSA	13	30 (3)	9.5	11 (3)
Llama	1	MSA	27	20 (1)	8	7.5 (1)
	2	MSA	27	32 (8)	14.5	10 (8)
	3	MSA	29	19 (7)	13	8 (7)
	4	MSA	30	30 (3)	11	11 (3)
Mistral	1	MSA	34	20 (1)	10.5	7.5 (1)
	2	MSA	35	32 (8)	7	10 (8)
	3	MSA	35	19 (7)	13	8 (7)
	4	MSA	35	30 (3)	10	11 (3)
Qwen	1	MSA	6	20 (1)	2	7.5 (1)
	2	MSA	29	32 (8)	12.5	10 (8)
	3	MSA	28	19 (7)	12.5	8 (7)
	4	MSA	28	30 (3)	8.5	11 (3)

Architectural style. Notably, in only two of 24 runs, the LLMs chose an architectural style other than Microservices. Except for Scenario 4, *GPT* decided on the same style as most participants from the study also did. *Code* matched with the opinion of the participants for the architectural style in Scenarios 2 and 3, which is the same for *DS*, *Llama*, *Mistral*, and *Qwen*. As described in Sect. III, the desired architectural style for Scenario 1 is Monolithic, for Scenario 2 Microservices, and Scenarios 3 and 4 are open for Monolithic, Server/Client, and Microservices.

Number of pattern. Table III compares the number of patterns that CAPI suggests. *GPT* suggested more patterns than the participants in Scenarios 1, 3, and 4, and fewer in Scenario 2. In Scenario 3, the list of suggested patterns is notably larger than that of the participants. *Code* and *Mistral* produced more pattern suggestions for all scenarios than the participants. For both *Code* and *Mistral*, it is notable that for Scenarios 1 and 3, there are particularly many more patterns suggested. Moreover, *Mistral* got nearly the same number of patterns suggested for all four scenarios. In contrast to *Code* and *Mistral*, *DS* received fewer pattern suggestions for all scenarios than the participants. For Scenarios 2 and 4, this difference is particularly striking. With *Llama*, the differences were by Scenario 1 and 3, with more patterns, and Scenario 2 with fewer patterns. In Scenario 4, *Llama* received the same number of suggested patterns by CAPI as the participants, which is the only case where that appeared. Finally, *Qwen* received fewer patterns in Scenarios 1, 2, and 3, whereas in Scenario 1, the difference is at its strongest, and in Scenario 3, there are more patterns. In general, a higher number of suggested patterns means that a participant or an LLM has answered the contraindication questions with "no". As the initial intention of this question was to make sure patterns were only added if the user of CAPI really needs them, this could hint that the question design of the contraindication questions still could be improved. On the other hand, the LLMs mainly chose Microservices as the architectural style. In this case, the decision tree of the CAPI method suggests more patterns in general, as there are more patterns needed for a well-structured microservice-based architecture than for a Monolithic architecture. Also, the decision tree has more questions to ask in the Microservices branch compared to the branches of Monolithic or Server/Client, which potentially causes the CAPI method to suggest more patterns.

Comparing the scores of the LLMs with those of the study participants, it is striking that the overall distance score of the LLMs is higher. Only six of the 24 lists of suggested patterns are closer to our desired list of patterns, and three are equal to the participants. For *GPT*, the scores for all scenarios were higher, but for Scenarios 1 and 3 differences were more substantial. For *Code*, Scenarios 1, 3, and 4 are higher, and 2 is lower. However, only in Scenario 3, the difference between the LLM and the participants is substantially bigger. *DS* has equal results for Scenarios 1 and 3 and higher for 1 and 2, whereas the difference for 2 is very substantial. *Llama* received a higher score except for Scenario 4, where Scenarios 2 and

3 have a substantial difference and Scenario 4 has an equal score. *Mistral* has a higher score for Scenarios 1 and 3, where the difference for Scenario 3 is substantial. For Scenarios 2 and 4, *Mistral* has a smaller distance. Finally, *Qwen* has a higher score for Scenarios 2 and 3, where 3 is substantial, and for Scenarios 1 and 4, a smaller difference, where for Scenario 1, this difference is substantially smaller and the overall nearest score to our desired list of patterns.

C. Experiment 3: LLMs' tool suggestions

In the final experiment, we examined whether the LLMs can not only recommend architectural patterns but also suggest tools that implement them. Following the procedure outlined in Sect. IV, we started from the improved prompt used in Experiment 1 and added two extensions: (1) a requirement that the response be returned in a structured JSON format so that it can be consumed directly by downstream tools such as CAPI, and (2) a predefined list of candidate tools from which the model must choose. We show the results of this experiment in Table IV.

TABLE IV
RESULTS OF EXPERIMENT 3: LLM SUGGEST REASONABLE TOOLS.

Model	Scenario	Style	Structured answer	Reasonable
GPT	1	Microservices	x	x
	2	Microservices		x
	3	Microservices		x
	4	Microservices		x
Code	1	Microservices	x	x
	2	Microservices		x
	3	Microservices		x
	4	Microservices	x	x
DS	1	Monolithic	x	x
	2	Microservices	x	x
	3	Microservices	x	x
	4	Microservices	x	x
Llama	1	Microservices	x	x
	2	Microservices	x	x
	3	Microservices	x	x
	4	Microservices	x	x
Mistral	1	Microservices		x
	2	Microservices		x
	3	Microservices		x
	4	Microservices		x
Qwen	1	Microservices		x
	2	Microservices		x
	3	Microservices		x
	4	Microservices		x

We reused the four scenarios from Sect. III. Compared with Experiment 2, the only notable change in the chosen architectural style is that *GPT* now recommends a microservice architecture for Scenario 1 (instead of client-server). All LLMs produce a much shorter list of architectural patterns than the CAPI method, and the patterns they do suggest correspond to the categories in the tool list we supplied [13].

The tools paired with those patterns are consistently sensible, and every suggested pattern is accompanied by a concrete tool—no pattern is left without an implementation option. Only *DS* and *Llama* generated output that strictly follows the required JSON format. *GPT* succeeded for the first scenario only, and *Code* did so for scenarios 1 and 4. In all other cases the models returned markdown or added extraneous text. The severity of the format violations varies: *GPT*, *Qwen*, and *Mistral* wrap the JSON in a markdown code block. In contrast, *Code* ignores the constraint in Scenarios 2 and 3, producing an unstructured list or a mixture of JSON and YAML ("*Based on the requirements... .Here are some...patterns...*").

VI. KEY FINDINGS

In this section, we will present the key findings from our experiments and align them with our research questions.

A. RQ1: How well does an LLM perform in suggesting architectural patterns?

In general, except for *Qwen*, the LLMs performed very well even with a naïve prompt (Expt. 1). As the given scenarios already give a concrete setting in which the LLM should act, the overall reasonability of the answers is good. This behavior is also underlined by the fact that by using the improved prompt, the LLMs answered more in our desired direction. The key takeaway and answer to this research question is that most LLMs, even with a naïve prompt, can correctly analyze concrete scenarios and provide reasonable suggestions (Expt. 1). For more targeted analysis tasks, like providing *architectural* patterns, however, further techniques are needed (Expt. 1, Expt. 2). Although the suggested patterns in our experiments were a good starting point, a software architect should not consider the suggestions as a complete analysis for this desired application, as the number of suggested patterns strongly deviates between prompting and the usage of the CAPI method (Expt. 1, Expt. 2).

B. RQ2: Can the results of an LLM be improved if it uses a decision tree to suggest architectural patterns?

As mentioned before, the general ability of LLMs to suggest architectural patterns delivers a good starting point but misses many patterns (Expt. 1). With our second experiment, we could show that by using our CAPI method, the LLMs could increase the completeness of suggested patterns. Nevertheless, those results are still not complete, as the CAPI method may not cover all available architectural patterns. However, interestingly, compared with the participants from our study to improve the CAPI method, the LLMs performed nearly as well as our participants and, in some cases, even better (Expt. 2). Furthermore, the LLMs have a strong tendency towards the usage of microservices-based architectures. As microservices were strongly hyped in recent years, such designed applications may be more present in the training data of the models (Expt. 2). All in all, the key findings for this research question are that LLMs tend to suggest microservices-based applications with their corresponding architectural patterns, which may be overengineering for specific scenarios, such as our first one (Expt. 1). Furthermore, the usage of the CAPI method increases the accuracy of the LLMs by giving them a structured list of questions aiming to identify the

requirements for a scenario and therefore suggest the needed patterns (Expt. 1, Expt. 2).

C. RQ3: How well does an LLM derive concrete tools from architectural patterns?

As we learned from our other experiments, the quality of the results of the LLMs increases if we describe the task environment in detail (Expt. 1). As we have provided a list of possible tools that implement previously suggested patterns, all of our tested LLMs performed exceptionally well to suggest these tools from the given list (Expt. 3). However, as the tools were categorized instead of a plain list of tools, these categories influence the LLMs to choose other patterns than it does in the first experiment (Expt. 3). Also, compared to the number of suggested patterns from the second experiment, in the third experiment, the LLMs again suggest a smaller and incomplete list of patterns (Expt. 1, Expt. 2, Expt. 3). Furthermore, in the third experiment, the LLMs suggest microservice-based architecture, too. Also, *GPT* changed its mind for the first scenario in favor of Microservices. Unfortunately, the third experiment also shows that the direct integration of the LLMs with its ability to suggest tools for patterns is only possible with two of six models, as only *DS* and *Llama* were able to answer in a structured JSON format that does not need to be prepared to be readable by a parser. The key findings for this research question are that LLMs can easily suggest fitting tools for architectural patterns if they have a list of available tools (Expt. 3). However, if the list has any categorization, this may influence the pattern suggestions of the LLM, and not all LLMs can answer in plain JSON even if it is enforced through the prompt (Expt. 3).

VII. Conclusion

In this paper, we studied the capabilities of diverse LLMs to propose architectural patterns and concrete tooling for given scenarios. Addressing RQ1, we found that LLMs can suggest reasonable patterns even under naïve prompting, with prompt engineering primarily improving architectural-pattern specificity and answer consistency rather than overall reasonability. For RQ2, embedding our CAPI method increased pattern coverage, but models frequently defaulted to microservices and often produced higher distance scores than human participants. Concerning RQ3, LLMs reliably selected plausible tools from a predefined list, though category cues in the list influenced pattern choices, and only two models consistently adhered to a strict, parser-ready JSON format (others required light post-processing). Overall, our experiments demonstrate that LLMs, when guided by structured questionnaires and supported by curated prompts, can effectively augment architectural decision-making, while requiring tighter output control and bias mitigation. Future work includes expanding CAPI's pattern base, enforcing output formats through automated validation, and integrating LLM-driven recommendations into interactive design tools to streamline real-world software architecture workflows.

References

[1] A. Fan, B. Gokkaya *et al.*, "Large Language Models for Software Engineering: Survey and Open Problems," in *IEEE/ACM International Conference on Software Engineering: Future of Software Engineering*, 2023.

[2] X. Hou, Y. Zhao *et al.*, "Large Language Models for Software Engineering: A Systematic Literature Review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, pp. 220:1–220:79, 2024.

[3] M. Haoues, A. Sellami *et al.*, "A guideline for software architecture selection based on ISO 25010 quality related characteristics," *Int. J. Syst. Assur. Eng. Manag.*, vol. 8, no. 2s, pp. 886–909, 2017.

[4] A. Razzaq, "A Systematic Review on Software Architectures for IoT Systems and Future Direction to the Adoption of Microservices Architecture," *SN Comput. Sci.*, vol. 1, no. 6, p. 350, 2020.

[5] S. Farshidi and S. Jansen, "A Decision Support System for Pattern-Driven Software Architecture," in *Software Architecture*, H. Muccini, P. Avgeriou *et al.*, Eds., 2020.

[6] S. Copei, O. Hohlfeld, and J. Kosiol, "The (C)omprehensive (A)rchitecture (P)attern (I)ntegration method: Navigating the sea of technology," in *Software Architecture. ECSA 2025 Tracks and Workshops*, 2025, pp. 212–228.

[7] L. P. F. Guerra and N. Ernst, "Assessing LLMs for Front-end Software Architecture Knowledge," in *2025 IEEE/ACM International Workshop on Designing Software (Designing)*, 2025, pp. 6–10.

[8] J. Jahić and A. Sami, "State of Practice: LLMs in Software Engineering and Software Architecture," in *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*, 2024, pp. 311–318.

[9] 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 2024, Hyderabad, India, June 4–8, 2024*. IEEE, 2024, pp. 79–89.

[10] M. Soliman and J. Keim, "Do Large Language Models Contain Software Architectural Knowledge? : An Exploratory Case Study with GPT," in *22nd IEEE International Conference on Software Architecture*, 2025.

[11] R. Rubei, A. Di Salle, and A. Bucaioni, "LLM-Based Recommender Systems for Violation Resolutions in Continuous Architectural Conformance," in *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*, 2025, pp. 404–409.

[12] S. Copei, O. Hohlfeld, and J. Kosiol, "Industrial Views on DevOps Adoption Before and After Implementation: A Qualitative Comparison," in *Software Architecture. ECSA 2025 Tracks and Workshops*, 2026, pp. 202–211.

[13] S. Copei, J. Kosiol *et al.*, "Research bundle for "LLMs Choose the Right Stack: From Patterns to Tools"," 2025. [Online]. Available: https://zenodo.org/records/16890378

[14] L. Schmid, T. Hey *et al.*, "Software Architecture Meets LLMs: A Systematic Literature Review," *CoRR*, vol. abs/2505.16697, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2505.16697

[15] J. J. Maranhão and E. M. Guerra, "A Prompt Pattern Sequence Approach to Apply Generative AI in Assisting Software Architecture Decision-making," in *Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices*, 2024.

[16] S. K. Pandey, S. Chand *et al.*, "Design pattern recognition: a study of large language models," *Empir. Softw. Eng.*, vol. 30, no. 3, p. 69, 2025.

[17] R. Laue, J. a. J. Maranhão, and E. M. Guerra, "Asking ChatGPT for Pattern Recommendations: EuroPLoP 2024 Focus Group Report," in *Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices*, 2024.

[18] E. Gamma, R. Helm *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[19] S. Copei and J. Kosiol, "DevOps Patterns: A Rapid Review," in *Software Architecture. ECSA 2023 Tracks, Workshops, and Doctoral Symposium*, 2024, pp. 33–50.

[20] A. Doosthosseini, J. Decker *et al.*, "SAIA: A Seamless Slurm-Native Solution for HPC-Based Services," Jul. 2025. [Online]. Available: https://www.researchsquare.com/article/rs-6648693/v1

[21] S. Schulhoff, M. Ilie *et al.*, "The Prompt Report: A Systematic Survey of Prompting Techniques," *CoRR*, vol. abs/2406.06608, 2024.

[22] S. Ouyang, J. M. Zhang *et al.*, "An Empirical Study of the Non-Determinism of ChatGPT in Code Generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 2, 2025.