LLMs for Generation of Architectural Components: An Exploratory Empirical Study in the Serverless World

***Abstract*—Recently, the exponential growth in capability and pervasiveness of Large Language Models (LLMs) has led to significant work done in the field of code generation. However, this generation has been limited to code snippets. Going one step further, our desideratum is to automatically generate architectural components. This would not only speed up development time, but would also enable us to eventually completely skip the development phase, moving directly from design decisions to deployment. To this end, we conduct an exploratory study on the capability of LLMs to generate architectural components for Functions as a Service (FaaS), commonly known as serverless functions. The small size of their architectural components make this architectural style amenable for generation using current LLMs compared to other styles like monoliths and microservices. We perform the study by systematically selecting open source serverless repositories, masking a serverless function and utilizing state of the art LLMs provided with varying levels of context information about the overall system to generate the masked function. We evaluate correctness through existing tests present in the repositories and use metrics from the Software Engineering (SE) and Natural Language Processing (NLP) domains to evaluate code quality and the degree of similarity between human and LLM generated code respectively. Along with our findings, we also present a discussion on the path forward for using GenAI in architectural component generation.**

***Index Terms*—Architectural Component Generation, LLM, Serverless.**

I. Introduction

Ever since the inception of the field of Software Architecture (SA) [1], one of the goals has been to automate/semi-automate the generation of executable systems from architecture descriptions as this would increase compliance, promote traceability, etc. Over the years the SA community has been working towards realizing this goal through defining Architecture Description Languages (ADLs) and Domain Specific Languages (DSLs) [2]–[4] and further developing approaches for code generations using model transformations. However, their application to practice has been limited due to the steep learning curve, lack of extensibility, support for tooling, etc [3]. On the other hand, with recent advancements in AI, Large Language Models (LLMs) are moving us ever closer to a world of increased automation, with applications across multiple Software Engineering (SE) tasks, as described by Hou et al. [5]. They have been used for software development, maintenance, requirements engineering, and more, with code generation and program repair being the most common applications [5]. There have also been several commercial tools such as ChatGPT, GitHub Copilot, and the Cursor IDE. However, this code generation has been in the context of generating low level code snippets, with the generation of software architecture components using LLMs being an unexplored space.

To this end, we conduct an exploratory empirical study on the capability of LLMs to generate architectural components in the context of Functions-as-a-Service (FaaS), commonly referred to as serverless functions. FaaS supports event-driven architectural style and enables easy development due to the abstractions provided by the cloud provider, who manages the infrastructure for running the basic units of FaaS, called serverless functions. We choose serverless functions primarily due to the small size of their architectural component, as opposed to microservices or monoliths, where a single component may consist of thousands or even millions of lines of code. We believe that this can provide a first step when evaluating the architectural component generation capabilities of LLMs. We emphasize that the architectural components we deal with in our study are serverless functions, and we refer to the architectural component of FaaS as serverless functions in the remainder of this paper. As part of our study, we utilize 3 kinds of prompts containing information at different levels of abstraction, systematically select 4 open-source serverless repositories and 5 code-generation LLMs, generating a total of 145 serverless functions that we evaluate for functionality and code quality both with and without human intervention. The code and data for our study is available here [1].

The remainder of this paper is structured as follows: Section II provides background information about Serverless Functions, LLMs and some prompting methods. Section III describes related work. Section IV describes our research questions and the design of our study. Section V presents results, which are discussed in Section VI along with a look into a possible future for GenAI for Software Architecture and discusses the threats to validity of our study. Finally, Section VII presents our conclusion and future work.

II. Background

This section provides an overview of the concepts dealt with in this study, including serverless functions, LLMs for general use and for code generation and methods to improve performance of an LLM on a task.

[1]https://anonymous.4open.science/r/LLM-ComponentGen-62DE/README.md

A. Function-as-a-Service (FaaS) (Serverless Functions)

Function-as-a-Service (FaaS) is one of the most popular forms of the cloud computing paradigm called serverless computing [6]. It is widely adopted across domains due to its nature of being *ephemeral*, *event driven*, and *elastic*. Unlike traditional cloud computing paradigms, in FaaS, developers only write business logic in the form of fine-grained *functions*, which are deployed onto a cloud platform, where the cloud provider handles the hassle of managing and maintaining infrastructure, execution environment, which is abstracted away from the developer. FaaS functions are *event driven* as they only run when triggered by an event, such as a HTTP request, an update in a database, or a message arriving on a message queue. They are *ephemeral* since they only run for a short amount of time, after which they are descheduled by the cloud provider. They are also *elastic*, since they can scale automatically based on load. This also leads to potential cost savings by charging only for the time functions run. These properties and the multitude of languages supported makes FaaS applications quite easy to develop. Through slight abuse of notation, we refer to serverless functions and functions interchangeably in the remainder of this paper.

B. Large Language Models and Code Generation

Large Language Models (LLMs) are probabilistic machine learning models built on the Transformer architecture [7] that can mimic human language use. They are composed of billions of parameters and are trained on substantial content sourced from the internet. Though they can be trained on and used for other tasks, paradigmatic LLMs are *generative auto-regressive models*, meaning that they sequentially generate the next token in a sequence of tokens. Tokens are the units into which text is broken down when processed and generated by LLMs. They may correspond to an entire word or to a part of a word [8]. The maximum number of tokens that an LLM can process is called its *context length* or *context window*, and this limits the amount of information that can be provided in the input *prompt* by the user.

The coding ability of LLMs was improved by the creation of datasets like HumanEval [9] and Mostly Basic Python Problems (MBPP) [10] on which LLMs have been *fine-tuned*. Fine-tuning involves adapting a pre-trained model to perform better on a specific task.

C. Zero-Shot Prompting and Few-Shot Prompting

Unlike fine-tuning, zero-shot prompting and few-shot prompting do not require training of the language model or a dataset to fine-tune on. Zero-shot prompting involves directly making a model perform a task without illustrative examples, whereas few-shot prompting provides some examples in context. Few-shot prompting is also referred to as in-context learning, as it improves the performance of the model on a task without requiring updates to the model's parameters [11]. This significantly improves model performance on the task, especially for larger models, as shown by [12]. In this work, an instance of zero-shot prompting is to simply provide context (the content of which is discussed in IV-C6) and request a serverless function to be generated. For few-shot prompts, we additionally provide some serverless functions from the repository along with their description to the LLM and ask for the function corresponding to a description whose code is not specified.

III. Related Work

Over the years, a lot of promising work has been done in specifying software architecture using Architecture Description Languages (ADL) and further supporting the analysis and code generation of components using model transformations [3], [13]. In particular, an approach for the generation of source code in the Go programming language from π-ADL was proposed by Cavalcante et al. [14]. Further, an ADL for architecture modeling, code generation, and simulation of Cyber-Physical- Systems was proposed by Sharaf et al. [15], [16]. Bardaro et al. in their recent work [4] make use of AADL for modeling and further code generation in the robotics domain. However, to the best of our knowledge, not much work has been done in the serverless domain. Vladislav Tankov et al. [17] proposes Kotless which aims to simplify serverless development through the use of a DSL and a plugin to generate deployment code. However, the developer must still develop the application logic and code. There have been some works in using DSLs for supporting the specification and generation of microservices. Rademacher et al [18] propose an extensible approach to generate adaptable microservice code and deployment specifications from models developed using the modeling language, LEMMA. Further, Suljkanović et al. [19] proposed Silveria, a DSL that allows users to model the architecture of microservice-based systems and further generate executable code using model transformations. However as pointed out by Malavolta et al. [3], ADLs in general have a higher learning curve which often hinders practical adoption.

Significant work has been done on using LLMs for code generation, including [20]–[24]. There also exist several tools such as GitHub Copilot and the Cursor IDE. However, these focus on the generation of code snippets, such as classes or methods, and not at an architectural component level.

While Eskandani et al. [25] describe the application design of serverless systems using GenAI as an open research direction, it is through the lens of selecting an optimal pattern based on requirements. We seek to evaluate the capability of LLMs in generating architectural components by choosing serverless as the architectural style. To this end, we also conduct evaluations on the functionality, code quality, and similarity between human-written and generated code.

IV. Study Design

A. Goal

This study aims to evaluate the degree to which LLMs are able to generate software architecture components. The degree here refers to both the functional correctness and quality of code. Using the Goal-Question-Metric approach described in

[26], we formalize our goal to:
Analyze the effectiveness of LLMs
For the purpose of generating software architecture components
With respect to automatic software architectural component generation
From the viewpoint of software architects and developers
In the context of the Function-as-a-Service (FaaS) architectural style

B. Research Questions

RQ$_1$: *Can LLMs generate functional serverless functions?*

- **RQ**$_{1.1}$: *Can LLMs generate functional serverless functions without human intervention?*
- **RQ**$_{1.2}$: *Can LLMs generate functional serverless functions with minimal human intervention?*

In this RQ, we aim to evaluate whether the architectural components generated by LLMs are usable, to the extent that they satisfy tests defined in their containing repositories. To this end, we measure the outcome both when we directly use the generated serverless function, and after fixing minor errors described in Table III.

RQ$_2$: *How does the code quality of LLM-written serverless functions compare to human-written code?*
In this RQ, we move beyond functionality to evaluate the code quality of the architectural component generated by the LLM. We are also interested in measuring how similar generated serverless functions are to their human written counterparts.

C. Experiment Workflow

In this subsection, we first explain Function Masking, an adaptation of Masked Language Modeling from [27], which we use to generate architectural components. Then, we describe each stage of the workflow shown in Figure 1.

1) Function Masking: BERT [27] was trained on the Masked Language Modeling (MLM) task, where tokens were randomly masked from the input, with the objective to predict the masked token based on its context (both to the left and right). This task is useful and allows for self-supervised learning [28]. For example, in the sentence "The tall boy goes to the market", if "boy" is chosen to be masked, the model sees the input "The tall [MASK] goes to the market", where "[MASK]" is the special mask token, and must output "boy". We extend this concept to the serverless world through Function Masking. Given a serverless repository with multiple serverless functions, we mask one function and use the LLM to recreate the masked function, using different types and amounts of context, as explained in IV-C5.

2) LLM Selection: An extensive number of LLMs are available for use, and we select some for our study using leaderboards published in literature. We require two LLMs, one which will be used for summarizing the existing codebase, as described in IV-C5, and one for generating the new serverless function, as described in IV-C6. The reason for choosing two distinct models is two-fold:
Primarily, the tasks these two models will be performing are different. The summarization model should be able to create an abstractive summary from the code given to it as context along with capturing the architectural information. This necessitates that it has a long context length, so that the input prompt can accommodate all the code needed to create the summary. Since a dedicated benchmark does not exist for this task, we use the ChatBot Arena[2] [29], which evaluates human preference.
However, the code generation model performs a coding task for which there are existing benchmarks (such as HumanEval [9] and MBPP [10]) and fine-tuned models (such as DeepSeek-Coder, CodeQwen). We select models from the top 10 of the EvalPlus leaderboard[3] [30], while ensuring diversity across size and availability. Both EvalPlus and Chat-Bot Arena are evolving leaderboards, and the models selected were in the top 10 at the time of conducting the study.
Secondly, using the same model may propagate biases, such as what part of the codebase is considered important. This avoids the possibility of the model creating a summary that can only be deciphered by itself, but also understood by other models and humans.

We selected Gemini-1.5-Pro [31] for codebase summarization. The models selected for code generation are described in Table I below. Refer II-B for an explanation on "Number of Parameters" and "Context Window Size". "Availability" indicates whether the model can be hosted on-premise (Local) or if an API call to an externally managed server is needed (API). For those that offer both, we highlight the method we used for the study in bold. "License Type" refers to whether the model weights are publicly available (Open) or not (Proprietary).

Model Name	Number of Parameters	Context Window Size (in tokens)	Availability	License Type
Artigenz-Coder-DS-6.7B	6.7B	16,384	Local/**API**	Open
CodeQwen1.5-7B-Chat	7B	64K	Local/**API**	Open
DeepSeek-V2.5	236B	128K	Local/**API**	Open
GPT-3.5-Turbo	Unknown	4,096	API	Proprietary
GPT-4	Unknown	8,192	API	Proprietary

TABLE I: Selected Models for Code Generation

3) Repository Selection: To select open-source repositories using the serverless architectural style, we utilize the Wonderless [32] and AWSomePy [33] datasets. The Wonderless dataset is a collection of 1,877 serverless applications from GitHub in various languages. 72.2% of the repositories included are in JavaScript or TypeScript, 19% in Python, and 2.7% in Java. The AWSomePy dataset consists of 145 AWS Lambda based serverless functions written in Python. We aim to choose at least one repository in JavaScript, TypeScript and Python each as they make up almost 70% of runtimes used on AWS Lambda [4]. In our selection, we also seek to ensure that the repository is not a demo or toy application, by filtering based on number of GitHub stars and manual inspection. Finally, we desire repositories that have tests defined with

[2]Leaderboard available at https://lmarena.ai/
[3]Leaderboard available at https://evalplus.github.io/leaderboard.html
[4]https://www.datadoghq.com/state-of-serverless/

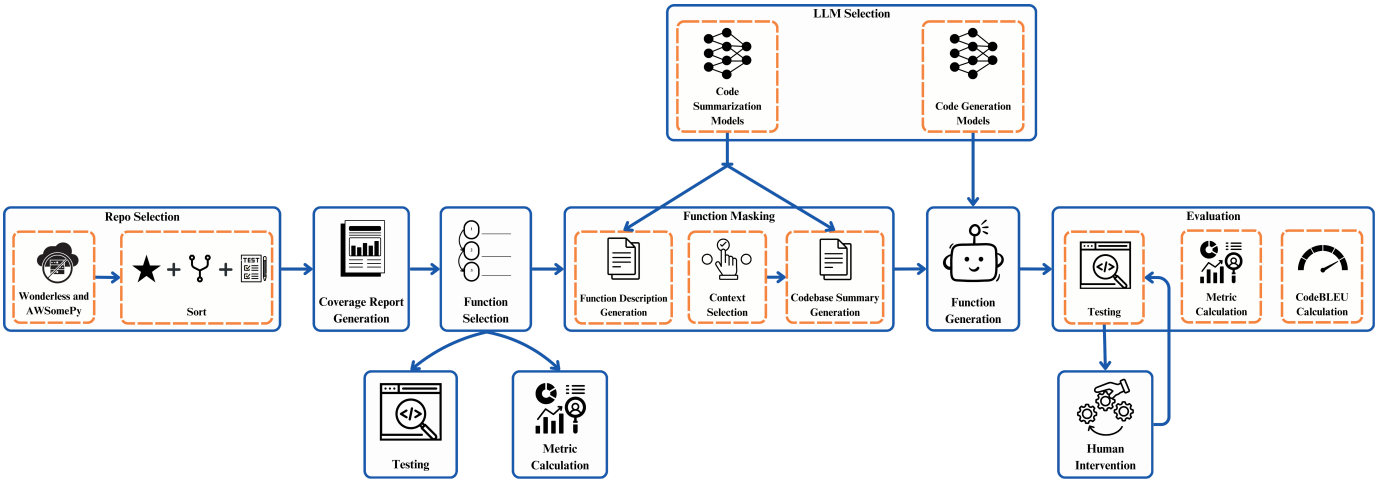

Fig. 1: Study Design

high test coverage to perform evaluations related to RQ_1, as described in IV-C7.

First, we sort the datasets based on GitHub stars and forks in decreasing order, to prioritize popular real world repositories. We then look for repositories that contain tests. As a first step heuristic, for every repository, we look for files or folders with "test" as a substring of their name. Then, for repositories with ≥ 30 stars on GitHub, we manually evaluate the quantity and quality of tests present. We found that some repositories had test files that were empty or too trivial to justify their selection. Additionally, several repositories contained serverless functions as only a small part of the entire codebase, usually for testing, and did not have tests for them. We also reject repositories that have been archived, since they may be deprecated or no longer relevant or maintained. For those repositories that had an acceptable number and quality of testcases, we calculated the test coverage and results. Based on these results, we selected four repositories. We describe the selected repositories in Table II below. All the repositories selected were from the Wonderless dataset.

Repository Name	Language	Stars	Forks	No. of Functions
codebox-npm	Javascript	352	27	10
laconia	Javascript	326	30	15
TagBot	Python	91	18	2
StackJanitor	Typescript	37	2	5

TABLE II: Selected Repositories

4) Function Selection: For each repository, we selected up to 3 functions to mask so that they can be generated by the LLM, enabling us to assess the model's performance across varying levels of complexity, length, and structure. We desire functions that have tests associated with them to enable the evaluation of the generated code. To this end, for each selected repository, we calculated the test coverage using the testing framework used in the repository. We then selected the functions with the highest statement coverage, using source lines of code as a tiebreaker. We finally select 10 serverless functions for masking across the 4 repositories.

5) Context Selection: In our study, we experimented with different levels of detail in the context provided to the LLM for code generation, focusing on two primary sources: the README file and a summary of the codebase which includes the architectural information.

For the README-based context, we used the repository's README file, which typically offers a concise overview, including a brief description, functionalities, and usage instructions. However, since README files often lack implementation details, this approach evaluated the LLM's ability to generate code with minimal context.

For the codebase summary-based context, we masked the target serverless function to generate a detailed codebase summary that excluded it. Using Gemini-1.5-Pro, as mentioned in Section IV-C2, we captured the architectural structure and functionality by extracting all other functions, their paths, and corresponding code (shown in IV-C5). This structured summary, enhanced with keywords like "components," "connectors," and "relationships" [27], provided context for evaluating the LLM's ability to reconstruct the masked function accurately.

Prompt for Codebase Summarization

Role Definition
Task Definition
Here is the codebase, with the path name of the file followed by the contents of the file in triple backticks:
{*FUNCTION PATH 1*}
```
{FUNCTION CODE 1}
```
⋮

Additionally, masking a function involves generating its function description (shown in IV-C5), which is later incorporated in the prompt for the LLM to generate the function from its description. We used Gemini-1.5-Pro to produce this function description from the function code. The full prompts can be viewed in our replication package.

Prompt for Function Description Generation

Role Definition
Task Definition
The function path and the function code itself (enclosed in triple backticks) are provided below:
{*MASKED FUNCTION PATH*}
```
{*MASKED FUNCTION CODE*}
```

6) Generation Methods: We follow multiple generation methods through different prompts

Prompt Types: From the two kinds of context described above, we create three prompts for function generation:

1) *Zero Shot with README* (Type 1 Prompt): This prompt (as shown in 1) contains no examples of other serverless functions and their descriptions and only provides the README file of the repository as context. The description of the masked function is provided and the model is tasked with generating the code for the serverless function .

Zero Shot with README

Role Definition: You are a computer scientist specializing in serverless computing (especially FaaS).
You are working with a FaaS codebase whose README is as follows:
{*README*}

Task Definition
The function should have the following functionality:
{*FUNCTION DESCRIPTION*}
Detailed Instructions Including Formatting

2) *Zero Shot with Codebase Summarization* (Type 2 Prompt): This prompt (as shown in 2) also contains no examples of other functions and descriptions, but includes the architectural information through the summary of the codebase, along with the description of the masked serverless function.

Zero Shot with Codebase Summarization

Role Definition: You are a computer scientist specializing in serverless computing (especially FaaS).
You are working with a FaaS codebase whose README is as follows:
{*CODEBASE SUMMARY*}

Task Definition
The function should have the following functionality:
{*FUNCTION DESCRIPTION*}
Detailed Instructions Including Formatting

3) *Few Shot with Codebase Summarization* (Type 3 Prompt): Along with the architectural information through the summary of the codebase, this prompt (as shown in IV-C6) also contains descriptions and function code of other serverless functions in the repository. These serve as guides for the model to generate the code for the masked function from the given description. This few-shot prompt allows the LLM to learn in-context without modifying model parameters, as described in Section II-C.

Few Shot with Codebase Summarization

Role Definition: You are a computer scientist specializing in serverless computing (especially FaaS). You are working with a FaaS codebase whose README is as follows:
{*CODEBASE SUMMARY*}

Task Definition
Here are some examples.
{*EXAMPLE FUNCTION DESCRIPTION 1*}
```
{*EXAMPLE FUNCTION CODE 1*}
```
{*EXAMPLE FUNCTION DESCRIPTION 2*}
```
{*EXAMPLE FUNCTION CODE 2*}
```

The function you generate should have the following functionality:
{*FUNCTION DESCRIPTION*}
Detailed Instructions Including Formatting

The full prompts can be found in our replication package. When making the prompts, we made an effort to be as detailed and specific as possible.

Consistency Check: LLMs are inherently probabilistic, and can produce a different output even with the same prompt. In the context of our work, this can result in the generated code passing tests and being of good quality in one run, but not in the other. To alleviate this issue, we first verify if multiple generations using the same context generate similar code. We compare code similarity using CodeBLEU [34], which is a weighted combination of n-gram matches[5], syntactic matches in the Abstract Syntax Trees (ASTs) and semantic data-flow matches. The results are shown in Figure 2. We see that generated functions are quite similar to each other for a model, and conclusions drawn by evaluating one function generated by a model will hold even when the model is given multiple tries. We finally generate 145 serverless functions for evaluation.

7) Evaluation Metrics: We perform three kinds of evaluations on the LLM generated serverless functions:

Functional Correctness Through Testing: To address RQ_1, we evaluated both the original and generated code using the

[5]an n-gram match is n tokens matching in order between the input and the output

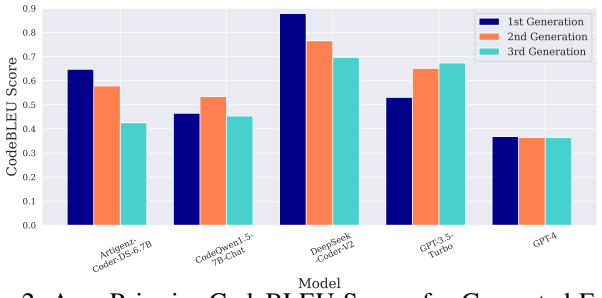

Fig. 2: Avg. Pairwise CodeBLEU Scores for Generated Functions per Model

existing tests in each repository. Specifically, we recorded the number of passing and failing tests for the entire codebase as well as for the individual functions selected for generation. After generating each function, we re-ran the tests and recorded the updated counts of passing and failing cases.

To address $RQ_{1.1}$, we initially conducted this evaluation without any code modifications. Here, we used the generated response of the LLM, cleaned to include only the code. Although this process was done manually, it can be automated by extracting only the text within triple backticks (```), as the model was instructed to provide code in this format. However, manual cleaning was necessary for cases where the model did not follow the formatting instructions.

To address $RQ_{1.2}$, we identified and fixed code generation errors made by LLMs. Song et al. [35] study the errors in LLM generated code and create a taxonomy of observed errors. We filter this to only include those errors that were frequently observed and solvable with minimal human intervention, shown in Table III. After correcting one or more of these issues, we recorded the updated counts of passing and failing tests, both for the overall codebase and for individual functions.

Error Categories	Error Types	
Semantic Errors	Condition Error	Missing condition Incorrect condition
	Constant Value Error	Constant value error
	Reference Error	Wrong method/variable Undefined name
	Calculation Error	Incorrect arithmetic/ comparison operation
	Incomplete Code	Missing one statement
	Memory Error	Infinite loop Integer overflow
Syntactic Errors	Conditional Error	If error
	Loop Error	For/ While error
	Return Error	Incorrect return value
	Method Call Error	Incorrect function name/ arguments Incorrect method call target
	Assignment Error	Incorrect arithmetic Incorrect constant Incorrect variable name Incorrect comparison
	Import Error	Import error

TABLE III: Errors Identified and Fixed in Generated Function

Code Quality through Code Metrics: Though the component generated may be functional, in the sense that it passes tests, we also desire code that is of good quality, to ensure quality attributes such as maintainability, readability and extensibility. In addressing RQ_2, we quantify code quality using code level metrics. Our choice for metrics is guided by previous work by Jin et al. [36], who analyze Java, JavaScript, and Python repositories. The popular Chidamber & Kemerer object-oriented metrics [37] are not applicable, since serverless functions written in the languages studied in this work need not be object-oriented. This is further supported by Jin et al. [36], who also avoid using OO metrics in similar contexts. We calculate the following metrics for the original and generated serverless functions:

1) Source Lines of Code (SLOC) - SLOC quantifies the size of a program by counting the number of lines which contain source code.
2) Halstead's Software Science Metric (Volume) [38] - Halstead's volume is another measure to quantify the size of a program which considers programs as a collection of tokens (operators and operands).
3) McCabe's Cyclomatic Complexity [39] - Cyclomatic complexity is used to measure the complexity of a program's control flow. It provides insight into the testability of the program, since one with a high cyclomatic complexity has more paths that need to be tested.
4) Cognitive Complexity [40] - This is a metric designed to measure the understandability of code.

Code Similarity using CodeBLEU: In answering RQ_2, we also measure how syntactically similar LLM generated serverless functions are to human written ones through the CodeBLEU [34] metric, which is described in IV-C6. A high CodeBLEU score indicates that the human-written and generated code are very similar, and the models may have appropriately created the summary and generated function. Though CodeBLEU measures semantic data-flow similarity, we utilize it to compare the syntactic similarity of serverless functions and use testing to evaluate semantic correctness. This is because the generated function may not pass tests despite a high CodeBLEU score with the original function due to subtle differences in the code that affects functionality or creates errors but does not significantly reduce CodeBLEU, such as incorrect imports or exchanged variables.

V. Results

A. RQ_1

1) $RQ_{1.1}$: Can LLMs generate functional serverless functions without human intervention?:
Table IV summarizes the percentage of test cases passed for the entire codebase and individual serverless functions, comparing original and LLM-generated functions across all models and prompt types. Type 3 prompts achieved the highest average test passing rates, with 73% of tests passed for the entire codebase. Comparing Type 1 prompt and Type 2 prompt for serverless function generation, the latter resulted in a higher test passing rates.

Figures 3(b-d) and (e-f) provide examples illustrating how the prompt types impact the generated function's structure and functionality. In Figure 3(b), generated by DeepSeek-Coder-V2 with the type 1 prompt, `findTag()` and `getStackJanitorStatus()` rely on hard-coded strings

Model Name	Repository Name (Language)	Percentage of Test Cases Passed (%)													
		Original		Generated											
				Type 1 Prompt				Type 2 Prompt				Type 3 Prompt			
			λ			λ				λ				λ	
		👤✗	👤✓	👤✗	👤✓	👤✗	👤✓	👤✗	👤✓	👤✗	👤✓	👤✗	👤✓	👤✗	👤✓
Artigenz-Coder-DS-6.7B	laconia (JS)	100	100	74.36	74.36	0	0	74.36	74.36	0	0	74.36	74.36	0	0
	codebox-npm (JS)	100	100	28.94	60.07	0	0	28.94	91.94	0	0	60.07	93.41	0	0
	StackJanitor (TS)	100	100	67.57	67.57	0	0	67.57	67.57	0	0	67.57	67.57	0	0
	Tagbot (PY)	93.33	100	0	0	0	0	82.67	82.67	0	0	-	-	-	-
CodeQwen1.5-7B-Chat	laconia (JS)	100	100	74.36	74.36	0	0	74.36	74.36	0	0	74.36	74.36	0	0
	codebox-npm (JS)	100	100	28.94	60.07	0	0	28.94	93.04	0	16.67	64.1	95.6	22.22	66.67
	StackJanitor (TS)	100	100	68.47	68.47	11.11	11.11	67.57	70.27	0	33.33	67.57	70.27	0	33.33
	Tagbot (PY)	93.33	100	0	0	0	0	82.67	86.67	0	37.5	-	-	-	-
DeepSeek-Coder-V2	laconia (JS)	100	100	74.36	74.36	0	0	74.36	74.36	0	0	74.36	79.49	0	66.67
	codebox-npm (JS)	100	100	31.87	91.94	0	0	91.94	94.51	0	41.67	94.14	95.6	38.89	66.67
	StackJanitor (TS)	100	100	67.57	81.08	0	33.33	67.57	90.09	0	62.5	75.68	95.5	0	79.17
	Tagbot (PY)	93.33	100	82.67	86.67	0	37.5	82.67	90.67	0	50	-	-	-	-
GPT-3.5-Turbo	laconia (JS)	100	100	74.36	74.36	0	0	74.36	76.92	0	33.33	74.36	94.87	0	62.5
	codebox-npm (JS)	100	100	60.81	91.94	0	0	60.81	94.87	0	55.56	60.81	95.6	11.11	66.67
	StackJanitor (TS)	100	100	67.57	78.38	0	0	67.57	76.58	0	62.5	67.57	76.58	0	62.5
	Tagbot (PY)	93.33	100	0	84	0	12.5	82.67	90.67	0	50	-	-	-	-
GPT-4	laconia (JS)	100	100	74.36	76.92	0	33.33	74.36	100	0	100	74.36	87.18	0	20.83
	codebox-npm (JS)	100	100	31.87	91.94	0	0	60.81	62.27	0	33.33	93.77	95.6	30.56	66.67
	StackJanitor (TS)	100	100	67.57	75.68	0	0	67.57	90.09	0	62.5	67.57	86.49	0	62.5
	Tagbot (PY)	93.33	100	82.67	86.67	0	37.5	82.67	90.67	0	50	-	-	-	-

TABLE IV: Test Case Pass Rates: for Codebase Tests, λ for Function Tests; 👤✗ (No Human Intervention) vs. 👤✓ (With Human Intervention)

(e.g., `"stackjanitor"` and `"enabled"`). This approach can reduce flexibility and misalign the function with actual codebase expectations. In contrast, Figure 3(c), generated by the same model with the type 2 prompt, avoids hard-coded strings by referencing a variable, showing an improvement. However, this version still lacks complete logic. Figure 3(d) shows a further improvement from the type 3 prompt, where `findTag()` is defined to return the tag object itself, which aligns closer to the original serverless function (in Figure 3(a)) and demonstrates the benefit of more detailed context.

Table IV reveals that larger models namely DeepSeek-Coder-V2, GPT-3.5-Turbo, and GPT-4 significantly outperformed smaller models (CodeQwen1.5-7B-Chat and Artigenz-Coder-DS-6.7B), regardless of prompt type. Notably, DeepSeek-Coder-V2 achieved the highest performance without human intervention, passing over 81% of codebase tests and 13% of individual function tests with type 3 prompt. GPT-4 followed with 78% and 10% test passing rates, respectively. In contrast, smaller models such as Artigenz-Coder-DS-6.7B and CodeQwen1.5-7B-Chat showed significantly lower passing rates. Figure 3(d) generated by DeepSeek-Coder-V2, and Figure 3(g) generated by Artigenz-Coder-DS-6.7B with the type 3 prompt, illustrate this difference - the former provides an accurate response in line with the original function, while the latter produces only a stub.

The choice of language also influenced the functionality, with javaScript-based repositories achieved higher pass rates in comparision to the other languages.

2) $RQ_{1.2}$: Can LLMs generate functional serverless functions with minimal human intervention?:

The results after applying minimal human intervention to the generated functions, as shown in Table IV, demonstrate a significant increase in test passing rates across all models and prompt types. DeepSeek-Coder-V2, which performed well with respect to RQ1.1, achieved over 90% success on codebase tests and 71% on individual function tests when using the Type 3 prompt. GPT-4 and GPT-3.5-Turbo also showed substantial performance improvements, with pass rates close to DeepSeek-Coder-V2. Even the lower-performing models, Artigenz-Coder-DS-6.7B and CodeQwen1.5-7B-Chat, saw marked improvements: their codebase test pass rates increased from 67% to 78% and 68% to 80%, respectively. However, their results still fell short of the larger models.

For instance, as illustrated in Figure 3(d) — an output generated by DeepSeek-Coder-V2 — correcting a return error based on the guidance in Table III, specifically by adjusting `findTag()` to return the entire tag object instead of just the tag value, improved the functionality of the generated code. Conversely, Figure 3(g) shows an output generated by Artigenz-Coder-DS-6.7B with the Type 3 prompt, which is a stub to benefit from minor corrections, highlighting the limitations of smaller models in producing functional code even with minimal human intervention.

B. RQ_2: How does the code quality of LLM-written serverless functions compare to human-written code?

From Figure 5, we see an overall reduction in the values of all code metrics except for some outliers in Cognitive Complexity for LLM generated serverless functions. For average SLOC, we see a reduction for all models and prompt types. Interestingly, we see that coding specific models generate more SLOC than the general purpose models, despite Artigenz-Coder-DS-6.7B and CodeQwen1.5-7B-Chat being much smaller than GPT-3.5-Turbo and GPT-4. Average Cyclomatic Complexity of the functions generated by the

```
export const findTag = (tags: CustomTag[]) =>
  tags.find((t) => t.key === TagName);

export const getStackJanitorStatus = (tags: CustomTag[]): StackStatus
  => {
  const tag = findTag(tags);
  if (tag && tag.value === StackStatus.Enabled) {
    return StackStatus.Enabled;
  }
  return StackStatus.Disabled;
};
```

(a)

```
const findTag = (tags: CustomTag[], tagKey: string): string {
    const tag = tags.find(tag => tag.Key === tagKey);
    return tag ? tag.Value : undefined;
};

const getStackJanitorStatus = (tags: CustomTag[]): StackStatus {
    const stackjanitorTagValue = findTag(tags, 'stackjanitor');
    return stackjanitorTagValue === 'enabled' ? 'Enabled' : 'Disabled';
};
```

(b)

```
function findTag(tags: CustomTag[], tagName: TagName): string => {
    const tag = tags.find(tag => tag.Key === tagName);
    return tag ? tag.Value : undefined;
}

function getStackJanitorStatus(tags: CustomTag[]): StackStatus => {
    const stackjanitorTagValue = findTag(tags, TagName.StackJanitor);
    return stackjanitorTagValue === "enabled" ? StackStatus.Enabled :
 StackStatus.Disabled;
}
```

(c)

```
const findTag = (tags: CustomTag[], tagName: TagName): CustomTag => {
  return tags.find(tag => tag.Key === tagName);
};

const getStackJanitorStatus = (tags: CustomTag[]): StackStatus => {
  const stackjanitorTag = findTag(tags, TagName.StackJanitor);
  return stackjanitorTag && stackjanitorTag.Value === "enabled" ? StackStatus.Enabled
  : StackStatus.Disabled;
};
```

(d)

```
import { getStackJanitorStatus } from './utils';
```

(e)

```
import { getStackJanitorStatus } from './helperFunctions';
```

(f)

```
function findTag(tagKey: string, tags: CustomTag[]): CustomTag | null {
  // Assume this function exists and finds a tag by its key in the tags array
}

function getStackJanitorStatus(tags: CustomTag[]): StackStatus {
  // Assume this function exists and gets the stackjanitor status from the tags array
}
```

(g)

Fig. 3: Snippets from Functions: (a) Original, (b-d) DeepSeek-Coder-V2 with Type 1-3 prompts, (e-g) Artigenz-Coder-DS-6.7B with Type 1-3 prompts.

models is also lower than the original, with the same trend of coding specific models demonstrating higher values despite smaller size. We observe that CodeQwen-1.5-7B-Chat and GPT-4 generated functions with higher Cognitive Complexity than the original functions, while Artigenz-Coder-DS-6.7B, DeepSeekCoder-V2 and GPT-3.5-Turbo have lower values than the original functions, with GPT-3.5-Turbo generating the most understandable code. Halstead volume of generated functions were much lower than the original functions across all models.

Regarding the difference in code quality that the different amount of context provided in the prompt to the models make, we find no appreciable difference in average SLOC and average Halstead Volume. However, we see a marked reduction in average cyclomatic complexity and average cognitive complexity when more context is provided to the model, with the lowest values demonstrated by the Few Shot Prompt with Codebase Summarization.

On comparing the similarity of human-written and LLM-generated functions using CodeBLEU score, we see from Figure 4 that Type 3 prompts make all models generate functions that are much more similar to the human written functions. We also see a correlation where larger models generate code more similar to the originals, with DeepSeek-Coder-V2 having the highest similarity score. Interestingly, in many cases, providing the codebase summarization in Type 2 prompt reduced CodeBLEU score compared to using the Type 1 prompt, though the former produced functions that passed more tests.

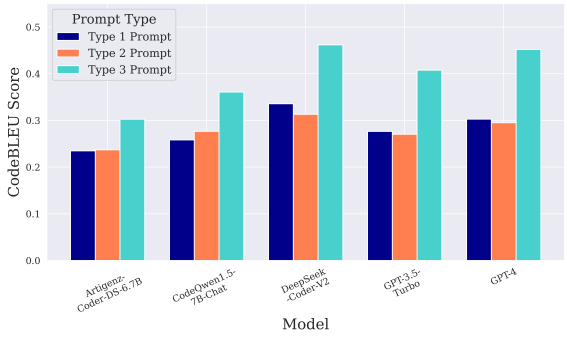

Fig. 4: Avg. CodeBLEU Scores for Generated Functions per Model and Prompt Type

VI. Discussion

In this section, we discuss lessons learnt from our study and threats to validity.

A. *Can LLMs generate functional serverless functions? (RQ_1)*

Yes, LLMs can be used to generate functional serverless functions with varying performance based on model and prompt type, but not entirely autonomously.

We observed that different levels of detail in the context provided to the LLM for serverless function generation significantly impact its performance. The Type 3 prompt produces the most functional serverless functions. This prompt type enabled better understanding of codebase structure and interdependencies, which was particularly beneficial to the larger models - DeepSeek-Coder-V2, GPT-4, and GPT-3.5-Turbo. This underlines the importance of both model scale and context in generating functional serverless functions. Smaller models - Artigenz-Coder-DS-6.7B and CodeQwen1.5-7B-Chat, constrained by their size, tend to generalize more, reducing the precision needed for function generation.

Interestingly, with respect to RQ1.2, involving minimal human intervention, the smaller models required substantial corrections, while the larger models needed relatively minor adjustments. However, in certain cases, the improvement in the functionality of the generated code was more pronounced for GPT-3.5-Turbo than GPT-4 which led GPT-3.5-Turbo to outperform GPT-4. Upon manual inspection of the generated functions, we found that GPT-3.5-Turbo tends to produce simpler initial outputs which, although often lacking full functionality, are generally easier to refine than GPT-4's more complex responses. GPT-4, while thorough in adhering closely

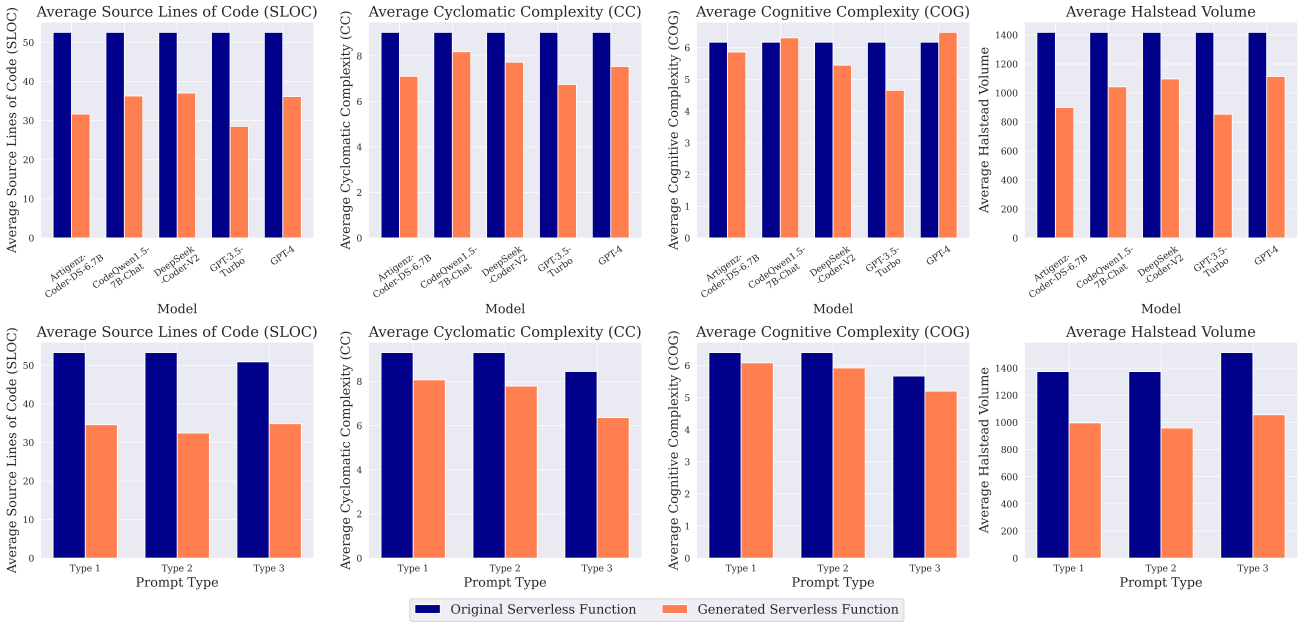

Fig. 5: Code Quality Metrics of Original and Generated Functions per Model (first row), per Prompt Type (second row)

to function requirements, occasionally produces code that is more challenging to adjust with minor changes.

As observed in V-A, JS-based repositories achieving higher pass rates, likely due to its prominence in serverless computing. This insight is valuable when selecting LLMs for architectural component generation.

Overall, while LLMs, especially the larger models, showcase the potential to generate functional serverless functions with minimal human intervention, there remains a gap between generated and fully functional human-written architectural components. This necessitates further research on leveraging LLMs to reach human-level proficiency in generating serverless functions autonomously, with an emphasis on maintaining a human in the loop to refine functionality, handle complex logic, and ensure practical applicability.

Main Findings for RQ1: LLMs can generate functional serverless functions with varying performance based on model and prompt type, but not to the extent to which humans can. Nonetheless, this method can be employed to aid architectural component generation.

B. How does the code quality of LLM-written serverless functions compare to human-written code? (RQ_2)

LLM-generated serverless functions show a reduction in Source Lines of Code, Cyclomatic Complexity, and Halstead Volume compared to human-written code. This suggests that LLMs produce concise and less complex code that is easier to understand and maintain. However, this simplicity can sometimes come at the expense of functionality, as demonstrated in Figure 3(g), where essential logic was omitted, impacting functional accuracy. Code generation models like Artigenz-Coder-DS-6.7B and CodeQwen1.5-7B-Chat produced more SLOC and higher Cyclomatic Complexity than general-purpose models like GPT-4 and GPT-3.5-Turbo, despite their smaller size. Among the models, larger ones, particularly DeepSeekCoder-V2, achieved the highest CodeBLEU scores, highlighting the importance of model size and prompt detail.

When comparing prompt types, Type 3 prompt, which provides detailed architectural and functional context, significantly reduced Cyclomatic and Cognitive Complexity, leading to simpler, and more understandable code. It also produced high similarity between LLM-generated and human-written functions, as illustrated in Figures 3(a) and (d). Despite the improvements in complexity metrics, Type 3 prompts had minimal impact on SLOC and Halstead Volume, indicating that while context simplifies logic, it does not necessarily shorten code length. This observation highlights the nuanced influence of context in shaping code quality.

An interesting anomaly was observed with Type 2 prompts, which occasionally reduced CodeBLEU scores despite generating functions that passed more tests compared to Type 1. This suggests that functional accuracy doesn't always align with human-code similarity.

Overall, while LLMs generate simpler and more understandable code, their limited ability to handle complex logic autonomously points to a trade-off between code quality and functionality, necessitating further refinement to achieve the desired functionality.

Main Findings for RQ2: LLMs can generate good quality serverless functions compared to human-written code. Larger models, combined with prompts that capture architectural and functional context, produce simpler, more understandable, and functionally accurate code, albeit with some trade-offs in complexity and functionality.

C. Defining Software Architecture in the Age of GenAI

Over the years, numerous definitions of software architecture (SA) have been proposed over the years, including by Perry and Wolf [41], Garlan and Shaw [1] and Jansen and Bosch [42][6]. When one considers component generation as a black-box, Jansen and Bosch's definition of SA as a set of design decisions [42] is more pertinent However, when moving deeper, during generation, it is necessary to consider the *components* that already exist in the system, how they are *connected* and what *constraints* exist, which is Garlan and Shaw's definition [1]. As we move into an age where GenAI is increasingly used in SA, it is imperative to be able to move between these definitions depending on the level of abstraction used. The need of the hour is accurate and detailed documentation of these design decisions to enable LLMs to generate the components and connectors subject to the constraints defined in the design decisions. These design decisions can be complemented by organizational context, and all of this can in turn benefit from the usage of GenAI, such as for documenting Architectural Decision Records (ADRs) as proposed by Dhar et al. [43].

D. The Path Forward for GenAI in SA

Perhaps the biggest challenge we faced in conducting this study was finding high quality data. Despite using published datasets, we found the quality of tests and components to be mostly unsatisfactory. Machine learning models need voluminous high quality data [44], and while this data exists to an extent for language modeling and code snippet generation, it is much more scarce when considering architectural component generation. There currently do not exist architecture specific tasks and benchmarks to evaluate LLMs, probably due to the aforementioned data scarcity.

The next issue that needs to be solved is the specific approach to using LLMs for generating architectural components. Various methods like Chain-of-thought prompting [45], retrieval-augmented-generation [46], knowledge graphs [47] [48] exist, and agentic frameworks [49], [50] exist, but their effectiveness for software architecture specific tasks have not been explored. LLMs could also be seen as a solution to address the learning curve of ADLs/DSLs where natural language could be converted to an architectural model and further code could be generated using model transformations. However, addressing these problems requires deeper collaboration between the SA and NLP communities.

E. Threats to Validity

We follow the categorization provided by Wohlin et al. [51] and also provide brief explanation about efforts taken to mitigate the identified threats or why it is not possible, as suggested by Verdecchia et al. [52].

External Validity: Selection of LLMs used for creating the context in the form of the codebase summary and for generating the serverless function poses a threat to external validity. However, we systematically select diverse LLMs using peer-reviewed published leaderboards, namely ChatBot-Arena [29] and EvalPlus [30], with EvalPlus specifically evaluating the coding abilities of LLMs. An interesting threat to external validity can be contamination of LLMs [53], which is when LLMs are trained or fine-tuned on data that is later used to evaluate their performance, leading to unrealistically high scores. Since most LLMs are not open about the data they are trained on, we have no way to mitigate this issue, especially since the repositories listed in Wonderless and AWSomePy were released in the public domain on which LLMs could have been trained. Despite this possibility, as shown in the results in Section V, LLMs were not able to generate completely functional serverless functions.

Internal Validity: The effectiveness of the tests in the repositories selected can be a threat to internal validity. However, we manually selected repositories which had high test coverage to mitigate this. Selection of metrics presents another threat, since both code quality and similarity are quite abstract concepts. To address this, we used metrics commonly used in the SE and NLP communities for measuring the same.

Construct Validity: A threat to construct validity stems from the selection of repositories and serverless functions in them. However, we utilize published datasets - Wonderless [32] and AWSomePy [33] with verifiable methods of collection. We filter them on the number of stars, quantity and quality of tests and test coverage to avoid picking toy or demo projects and retain repositories and serverless functions that have real-world relevance.

VII. Conclusion and Future Work

This study seeks to empirically explore the capabilities of Large Language Models to automatically generate software architectural components. We do this in context of serverless functions which is an event-driven architectural stlye, due to the small size of their basic architectural unit. Using a range of models, including general-purpose models like GPT-4 and code generation models like DeepSeekCoder-V2, and diverse prompt types, we evaluate the generated architectural components for both functional correctness and code quality. We find that while LLMs often fail to generate fully functional serverless functions autonomously, they can serve as a valuable starting point, as minimal human intervention significantly improves their functionality, enabling them to pass more tests. We also observe that LLM generated serverless functions display better metrics related to code quality, such as cyclomatic complexity, and cognitive complexity. Furthermore, in its current state, one cannot, and probably should not completely remove the human from the component generation process, suggesting a human-centered GenAI approach, as put forth by the Copenhagen Manifesto [54].

Future work involves exploring other techniques for generation used at the code level, such as those mentioned in VI-D, including exploring Retrieval Augmented Generation, multi-agent frameworks, and other architectural styles such as microservices and monoliths with larger components.

[6]See https://insights.sei.cmu.edu/library/what-is-your-definition-of-software-architecture/ for more definitions

REFERENCES

[1] D. Garlan and M. Shaw, "An introduction to software architecture," in *Advances in software engineering and knowledge engineering*, pp. 1–39, World Scientific, 1993.

[2] A. Mehmood and D. N. Jawawi, "Aspect-oriented model-driven code generation: A systematic mapping study," *Information and Software Technology*, vol. 55, no. 2, pp. 395–411, 2013.

[3] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 869–891, 2012.

[4] G. Bardaro and M. Matteucci, "Modelling robot architectures with aadl," *Ada Lett.*, vol. 43, p. 59–63, Oct. 2023.

[5] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, 2023.

[6] J. Wen, Z. Chen, X. Jin, and X. Liu, "Rise of the planet of serverless computing: A systematic review," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–61, 2023.

[7] A. Vaswani, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.

[8] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (K. Erk and N. A. Smith, eds.), (Berlin, Germany), pp. 1715–1725, Association for Computational Linguistics, Aug. 2016.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[10] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[11] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd ed., 2024. Online manuscript released August 20, 2024.

[12] T. B. Brown, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[13] M. Brun, J. Delatour, and Y. Trinquet, "Code generation from aadl to a real-time operating system: An experimentation feedback on the use of model transformation," in *13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008)*, pp. 257–262, IEEE, 2008.

[14] E. Cavalcante, F. Oquendo, and T. Batista, "Architecture-based code generation: From π-adl architecture descriptions to implementations in the go language," in *Software Architecture: 8th European Conference, ECSA 2014, Vienna, Austria, August 25-29, 2014. Proceedings 8*, pp. 130–145, Springer, 2014.

[15] H. Muccini and M. Sharaf, "Caps: Architecture description of situational aware cyber physical systems," in *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 211–220, IEEE, 2017.

[16] M. Sharaf, H. Muccini, and M. Abughazala, "Aria: arduino code generation based on the caps," in *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, ECSA '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[17] V. Tankov, Y. Golubev, and T. Bryksin, "Kotless: A serverless framework for kotlin," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1110–1113, IEEE, 2019.

[18] F. Rademacher, J. Sorgalla, P. Wizenty, and S. Trebbau, "Towards an extensible approach for generative microservice development and deployment using lemma," in *European Conference on Software Architecture*, pp. 257–280, Springer, 2021.

[19] A. Suljkanović, B. Milosavljević, V. Inić, and I. Dejanović, "Developing microservice-based applications using the silvera domain-specific language," *Applied Sciences*, vol. 12, no. 13, p. 6679, 2022.

[20] P. Bareiß, B. Souza, M. d'Amorim, and M. Pradel, "Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code," *arXiv preprint arXiv:2206.01335*, 2022.

[21] L. Gong, J. Zhang, M. Wei, H. Zhang, and Z. Huang, "What is the intended usage context of this model? an exploratory study of pre-trained models on various model repositories," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–57, 2023.

[22] A. Chen, J. Scheurer, T. Korbak, J. A. Campos, J. S. Chan, S. R. Bowman, K. Cho, and E. Perez, "Improving code generation by training with natural language feedback," *arXiv preprint arXiv:2303.16749*, 2023.

[23] J. Wang and Y. Chen, "A review on code generation with llms: Application and evaluation," in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pp. 284–289, IEEE, 2023.

[24] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," 2024.

[25] N. Eskandani and G. Salvaneschi, "Towards ai for software systems," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pp. 79–84, 2024.

[26] V. R. B. G. Caldiera and H. D. Rombach, "The goal question metric approach," *Encyclopedia of software engineering*, pp. 528–532, 1994.

[27] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of NAACL-HLT*, pp. 4171–4186, 2019.

[28] X. Liu, F. Zhang, Z. Hou, L. Mian, Z. Wang, J. Zhang, and J. Tang, "Self-supervised learning: Generative or contrastive," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 1, pp. 857–876, 2023.

[29] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, B. Zhu, H. Zhang, M. Jordan, J. E. Gonzalez, *et al.*, "Chatbot arena: An open platform for evaluating llms by human preference," in *Forty-first International Conference on Machine Learning*, 2024.

[30] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[31] G. Team, P. Georgiev, V. I. Lei, R. Burnell, L. Bai, A. Gulati, G. Tanzer, D. Vincent, Z. Pan, S. Wang, *et al.*, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," *arXiv preprint arXiv:2403.05530*, 2024.

[32] N. Eskandani and G. Salvaneschi, "The wonderless dataset for serverless computing," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 565–569, IEEE, 2021.

[33] G. Raffa, J. B. Alis, D. O'Keeffe, and S. K. Dash, "Awsomepy: A dataset and characterization of serverless applications," in *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*, SESAME '23, (New York, NY, USA), p. 50–56, Association for Computing Machinery, 2023.

[34] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[35] D. Song, Z. Zhou, Z. Wang, Y. Huang, S. Chen, B. Kou, L. Ma, and T. Zhang, "An empirical study of code generation errors made by large language models," 2023.

[36] S. Jin, Z. Li, B. Chen, B. Zhu, and Y. Xia, "Software code quality measurement: Implications from metric distributions," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pp. 488–496, IEEE, 2023.

[37] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[38] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. USA: Elsevier Science Inc., 1977.

[39] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[40] G. A. Campbell, "Cognitive complexity: An overview and evaluation," in *Proceedings of the 2018 international conference on technical debt*, pp. 57–58, 2018.

[41] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, p. 40–52, Oct. 1992.

[42] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pp. 109–120, IEEE, 2005.

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

[44] A. Jain, H. Patel, L. Nagalapatti, N. Gupta, S. Mehta, S. Guttula, S. Mujumdar, S. Afzal, R. Sharma Mittal, and V. Munigala, "Overview and importance of data quality for machine learning tasks," in *Proceedings*

of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD '20, (New York, NY, USA), p. 3561–3562, Association for Computing Machinery, 2020.

[45] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24824–24837, 2022.

[46] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[47] S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, and X. Wu, "Unifying large language models and knowledge graphs: A roadmap," *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[48] I. Abdelaziz, J. Dolby, J. McCusker, and K. Srinivas, "A toolkit for generating code knowledge graphs," in *Proceedings of the 11th Knowledge Capture Conference*, pp. 137–144, 2021.

[49] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," *arXiv preprint arXiv:2312.13010*, 2023.

[50] M. Zhuge, C. Zhao, D. Ashley, W. Wang, D. Khizbullin, Y. Xiong, Z. Liu, E. Chang, R. Krishnamoorthi, Y. Tian, *et al.*, "Agent-as-a-judge: Evaluate agents with agents," *arXiv preprint arXiv:2410.10934*, 2024.

[51] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, *et al.*, *Experimentation in software engineering*, vol. 236. Springer, 2012.

[52] R. Verdecchia, E. Engström, P. Lago, P. Runeson, and Q. Song, "Threats to validity in software engineering research: A critical reflection," *Information and Software Technology*, vol. 164, p. 107329, 2023.

[53] Y. Dong, X. Jiang, H. Liu, Z. Jin, B. Gu, M. Yang, and G. Li, "Generalization or memorization: Data contamination and trustworthy evaluation for large language models," *arXiv preprint arXiv:2402.15938*, 2024.

[54] D. Russo, S. Baltes, N. van Berkel, P. Avgeriou, F. Calefato, B. Cabrero-Daniel, G. Catolino, J. Cito, N. Ernst, T. Fritz, H. Hata, R. Holmes, M. Izadi, F. Khomh, M. B. Kjærgaard, G. Liebel, A. L. Lafuente, S. Lambiase, W. Maalej, G. Murphy, N. B. Moe, G. O'Brien, E. Paja, M. Pezzè, J. S. Persson, R. Prikladnicki, P. Ralph, M. Robillard, T. R. Silva, K.-J. Stol, M.-A. Storey, V. Stray, P. Tell, C. Treude, and B. Vasilescu, "Generative ai in software engineering must be human-centered: The copenhagen manifesto," *Journal of Systems and Software*, vol. 216, p. 112115, 2024.