The Impact of Prompt Programming on Function-Level Code Generation

Citation for the original published paper (version of record):
Khojah, R., Gomes, F., Mohamad, M. et al (2025). The Impact of Prompt Programming on Function-Level Code Generation. IEEE Transactions on Software Engineering, 51(8): 2381-2395.
http://dx.doi.org/10.1109/TSE.2025.3587794

The Impact of Prompt Programming on Function-Level Code Generation

Ranim Khojah, Francisco Gomes de Oliveira Neto, Mazen Mohamad, *Member, IEEE*, and Philipp Leitner

***Abstract*—Large Language Models (LLMs) are increasingly used by software engineers for code generation. However, limitations of LLMs such as irrelevant or incorrect code have highlighted the need for prompt programming (or prompt engineering) where engineers apply specific prompt techniques (e.g., chain-of-thought or input-output examples) to improve the generated code. While some prompt techniques have been studied, the impact of different techniques — and their interactions — on code generation is still not fully understood. In this study, we introduce CodePromptEval, a dataset of 7072 prompts designed to evaluate five prompt techniques (few-shot, persona, chain-of-thought, function signature, list of packages) and their effect on the correctness, similarity, and quality of complete functions generated by three LLMs (GPT-4o, Llama3, and Mistral). Our findings show that while certain prompt techniques significantly influence the generated code, combining multiple techniques does not necessarily improve the outcome. Additionally, we observed a trade-off between correctness and quality when using prompt techniques. Our dataset and replication package enable future research on improving LLM-generated code and evaluating new prompt techniques.**

***Index Terms*—Large language models, prompt programming, code generation.**

I. Introduction

WITH the widespread adoption of Large Language Models (LLMs) in software engineering, researchers and practitioners have uncovered their significant potential, particularly for code-related tasks, such as code generation and completion [1], [2]. However, this adoption has also revealed several limitations of LLMs that can hinder developers' productivity [3] and cause frustrations [4], preventing them from fully leveraging the benefits of LLMs in their coding process. Such limitations are related to hallucinations, misunderstanding the intent or purpose of the code, or simply generating incorrect code [5].

These limitations are inherent to the design of LLMs, and are unlikely to "resolve themselves" entirely with future model generations. Therefore, researchers started proposing ways to mitigate these limitations by adapting how users interact with the LLMs. The interactions typically start with a natural language prompt that specifies what the LLM is expected to output. To ensure that LLM generates accurate, relevant, and high-quality outputs, users employ a structured approach to construct prompts, which is known as prompt programming.

To implement prompt programming, various prompt techniques can be used to guide the LLM on how to achieve the expected results [6], [7], [8]. For example, few-shot learning involves providing the LLM with a few input-output examples to guide the function logic, while adding context about the packages used can give the model additional information on what helper functions to use.

However, such prompt techniques were evaluated based on the output accuracy for natural language generation tasks [8], [9] and are not well-studied for code generation, more specifically, function synthesis (generating function-level code), which is one of the most common use cases among software engineers [3]. Furthermore, evaluating the accuracy of code generation is not sufficient, since other aspects of the code are important for software engineers, such as maintainability and adherence to best practices. Prompt techniques can also be combined [6], but to the best of our knowledge, no work evaluates the impact of multiple *interacting* prompt techniques in one prompt. For instance, whether applying a certain prompt technique can cancel out, hinder, or even enhance the impact of an existing prompt technique in the prompt.

Therefore, in this study, we design a full factorial experiment on five common prompt techniques for function generation along with all the possible combinations of these prompts, which sums up to 32 unique combinations of prompt techniques. To perform a comprehensive evaluation of the impact of different prompt techniques on code generation, we construct our dataset CodePromptEval which consists of 221 code-generation prompts from CoderEval [10], that we extend with 32 possible variations for each prompt (that is, combinations of prompt techniques). This results in a total of 7072 datapoints. We use CodePromptEval to generate functions with three popular LLMs (GPT-4o, Llama3, and Mistral), then evaluate the

Received 26 December 2024; revised 3 July 2025; accepted 7 July 2025. Date of publication 10 July 2025; date of current version 15 August 2025. Recommended for acceptance by C. Treude. *(Corresponding author: Ranim Khojah.)*

Ranim Khojah, Francisco Gomes de Oliveira Neto, and Philipp Leitner are with Chalmers University of Technology and University of Gothenburg, 417 56 Gothenburg, Sweden (e-mail: khojah@chalmers.se; francisco.gomes@cse.gu.se; philipp.leitner@chalmers.se).

Mazen Mohamad is with Chalmers University of Technology and University of Gothenburg, 417 56 Gothenburg, Sweden, and also with RISE Research Institutes of Sweden, 504 62 Borås, Sweden (e-mail: mazen.mohamad@ri.se).

This article has supplementary downloadable material available at https://doi.org/10.1109/TSE.2025.3587794, provided by the authors.

Digital Object Identifier 10.1109/TSE.2025.3587794

generated functions based on correctness, as well as quality and similarity to ground truth (e.g., in terms of naming style and structure). Particularly, we investigate the following research questions.

RQ 1*: How do different LLMs perform on CodePromptEval?* Initially, we study the performance of different current-generation LLMs (GPT-4o, Llama3, and Mistral) on our CodePromptEval dataset. We particularly look at the correctness of LLM-generated code as measured using existing test cases in CoderEval benchmark as a ground truth. We observe that the performance of all three evaluated LLMs is comparable, with a difference of around 5 percentage points between the best model, GPT-4o, and the worst, Mistral.

RQ 2*: To what extent do different prompting techniques (and combinations of them) impact the code generation of LLMs?* We now turn to the central research question of this paper. Using a full factorial experiment design, we compare how different prompt techniques (e.g., few-shot, providing a persona, etc.) impact the generated code in three dimensions: correctness, similarity to ground truth, and code quality.

> ***RQ 2.1****: How do prompt techniques impact the correctness of the code?*
>
> To evaluate correctness, we test the functions, then measure the Pass@k scores for each combination of prompt techniques. We also perform statistical tests to identify the (combinations of) prompt techniques that impact the test results. We found that including only a function signature or few-shot examples has a significant positive impact on correctness. We further observe that combining prompt techniques does not lead to significantly better results.
>
> ***RQ 2.2****: How do prompt techniques impact the similarity of the code to a human-written baseline?*
>
> We also study how similar generated solutions are to the (human-written) baseline. We find that including a persona, chain-of-thought, or signature increases the overall similarity to the baseline for some LLMs, while few-shot reduces only the lexical similarity. Note that generating code that is similar to an "expected" solution may be good or bad depending on context — on the one hand, code that is close to the baseline may be easy to fix even if it is not passing the test cases; on the other, "different" can be particularly valuable if the goal is to brainstorm approaches, e.g., if used in "exploration mode" [11].
>
> ***RQ 2.3****: How do prompt techniques impact the quality of the code?*
>
> Finally, we study code quality as measured through the presence of code smells and the (cyclomatic and cognitive) complexity of the code. We find that including a signature or few-shot examples leads to functions with higher complexity and more code smells. Interestingly, adding a relevant persona ("as a software developer who follows best coding practices ...") indeed has a small positive effect on the code quality, but at the expense of slightly lower correctness.

Overall, we conclude that the impact of prompt programming techniques is not dramatic for, at the time of writing, current-generation models. Most combinations of prompt techniques do not lead to statistically significant improvements (nor regressions) in correctness, similarity or quality. Providing type information for the function that is to be generated, either explicitly through a signature, or implicitly via few-shot examples, has the most clear effect. Some prompt techniques have a positive impact on correctness, and others on quality. However, the obvious idea of combining them usually improves neither.

II. Related Work

Existing research on LLMs in software engineering has shown the potential of LLMs to support software engineers in various tasks, including requirements elicitation, software testing and documentation [12], [13]. However, the main focus is directed towards code-related tasks [3]. This is also reflected in the interest among software organizations that, at the time of writing, leverage LLMs mostly for code generation, code completion and code summarization [14]. However, the increased adoption of LLMs for code-related tasks has unveiled risks and limitations, such as hallucinations, inaccuracies, and potential vulnerabilities [15], [16]. Researchers have proposed the concept of *prompt programming* (or prompt engineering) in order to minimize the model's limitations and trigger the LLM to output a more desirable response by using prompt techniques and provide relevant contextual information [6], [17].

Therefore, a new line of research emerged focusing on finding prompt techniques that can improve the performance of LLMs in various tasks. White et al. [6] propose different prompt patterns and techniques depending on the software-related task. However, the impact of these techniques on the LLM output can be unstable and inconsistent. Wang et al. [18] shows that prompt techniques can be sensitive to the specific task as well as the LLM (e.g., GPT-3.5 vs. GPT-4o). Other studies also show that the few-shot prompt technique [19] is effective, especially with the right structure [7], type [20] or order [21] of the examples (shots). Reynolds and McDonell [9] highlight how few-shot examples can hurt the performance of the model and limit its search for a plausible solution in translation tasks. Contrastingly, we found that few-shot significantly improved the performance of the LLMs suggesting that prompt techniques have varied impact depending on the task and the domain.

For code-related tasks, prompt techniques were shown to have a positive effect on code generation in the domain of education [22]. Furthermore, researchers proposed ways and contextual information as prompt techniques to apply to the prompt and enhance code-related tasks [8], [23], e.g., incorporating dataflow information to improve code summarization [8]. Other prompt techniques used by Dong et al. [24] included self-collaboration, where the LLM is prompted several times to take different personas e.g., first as a requirements engineer,

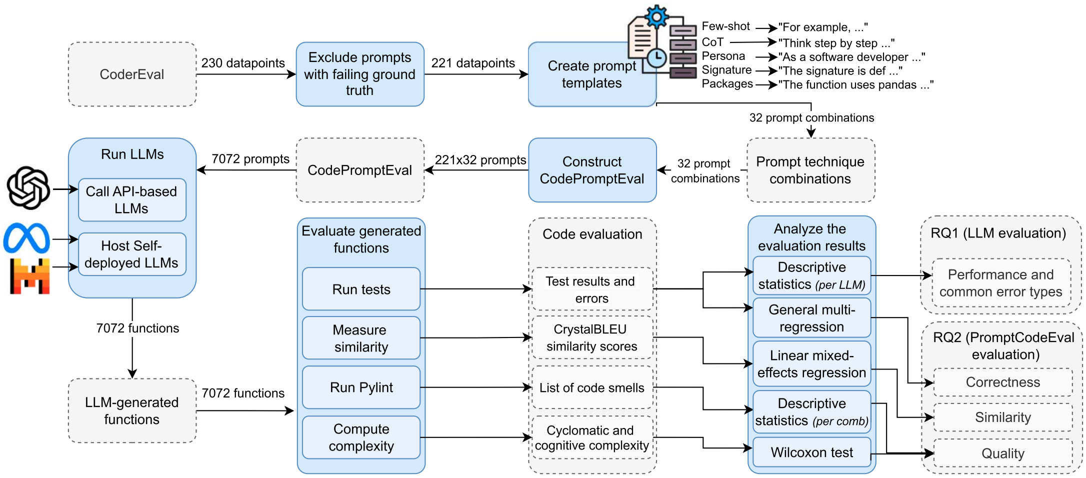

CoderEval
230 datapoints
Exclude prompts with failing ground truth
221 datapoints
Create prompt templates
Few-shot → "For example, ..."
CoT → "Think step by step ..."
Persona → "As a software developer ..."
Signature → "The signature is def ..."
Packages → "The function uses pandas ..."
32 prompt combinations
Prompt technique combinations
32 prompt combinations
Construct CodePromptEval
221x32 prompts
CodePromptEval
7072 prompts
Run LLMs
Call API-based LLMs
Host Self-deployed LLMs
7072 functions
LLM-generated functions
7072 functions
Evaluate generated functions
Run tests
Measure similarity
Run Pylint
Compute complexity
Code evaluation
Test results and errors
CrystalBLEU similarity scores
List of code smells
Cyclomatic and cognitive complexity
Analyze the evaluation results
Descriptive statistics (per LLM)
General multi-regression
Linear mixed-effects regression
Descriptive statistics (per comb)
Wilcoxon test
RQ1 (LLM evaluation)
Performance and common error types
RQ2 (PromptCodeEval evaluation)
Correctness
Similarity
Quality

Fig. 1. The process we follow to evaluate the code generated using different prompts by different LLMs (per run).

then a software developer, then a tester, and only then return a code that resulted from the "collaboration" among the three personas. Fagadau et al. [25] examined how individual prompt features (active voice and edge cases) affect code generation. Their experiments, which combined different prompt features, showed that most had little impact on the resulting code. In our study, we focus on three common prompt techniques, namely, few-shot learning, chain-of-thought [26], and persona [27], as well as propose two pieces of contextual information as additional prompt techniques that are easily accessed by the developers, i.e., the imported packages and the signature of the function. In addition, we investigate not only the impact of individual prompt techniques but also their interaction effects on code generation, for example, whether the effect of combining few-shot and persona arises from their interaction or from one technique alone.

To evaluate LLMs on code generation tasks, the most common metric is Pass@k [28], where k = 1 is used to measure the rate of passed functions that the LLM generated on the first attempt [24] (e.g., by running a test suite). CodeBLEU [29] is another popular metric, commonly used in studies to measure the human-likenesses of generated code [30], [31]. Li et al. [32] conduct a manual human evaluation of their proposed prompt technique "AceCoder" based on correctness, presence of code smells, and maintainability. We provide a systematic and automated approach to evaluate generated code based on correctness, maintainability, and similarity to the ground truth.

III. Methodology

Fig. 1 shows our approach to evaluate the impact of commonly-used prompting techniques on the code generated by LLMs. On a high level, we create prompt templates that combine prompting techniques (e.g., CoT, few-shot, etc.) and apply each prompt template to 221 tasks from the CoderEval benchmark [10]. We evaluate three different LLMs (two open-weight and one proprietary), leading to 7072 generated functions per LLM. To understand the impact of each prompting technique and answer RQ2, we evaluate all functions in terms of correctness, similarity to the baseline of the benchmark, and quality using statistical analysis.

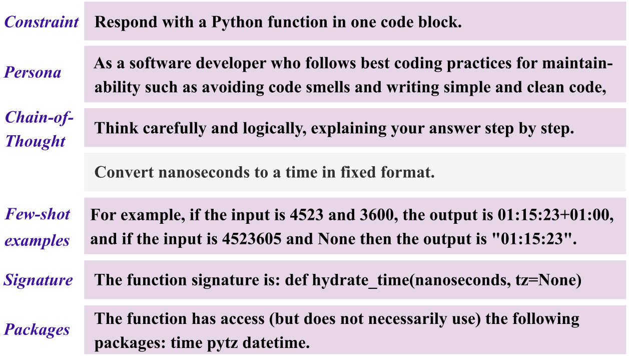
We follow a full factorial experiment design and evaluate the code generation functionality of LLMs by varying two levels (present/absent) of five factors in a prompt, that is, the five prompt techniques: (1) few-shot learning, (2) Chain-of-Thought (CoT), (3) persona, (4) function signature, and (5) the list of packages. Therefore, we have 32 (2^5) treatments in our experiment. Note that the absence of all of these techniques counts as zero-shot, where only the generation instruction is present without any other prompt technique. We do not treat zero-shot as a factor since it cannot logically be combined with other prompt techniques (e.g., combining zero-shot with persona would simply default to persona). Instead, we use zero-shot as a baseline for prompt technique comparisons.

A. Prompt Technique Combinations

Prompt programming is the act of constructing a prompt using natural language to ensure that the model provides the intended response or to improve the performance of the model [9]. Based on observations from our previous work [3] and recommendations in literature [6] and from LLM providers such as OpenAI[1] and Microsoft[2][3] we decided on five prompt techniques to apply when prompting LLMs in our study. Examples for all prompt techniques will be provided later in Fig. 2.

- *Few-shot learning* can be achieved by providing shots (or examples) to an LLM in order to enable learning new examples without the need to fine-tune the LLM [19]. Typically, the examples describe the structure of the input and output. In code generation, such pattern will result in each example being composed of a generation task in natural language as an input, and a complete function as an output. We found this to be impractical from a user perspective, and poses a challenge of what generation tasks to choose in terms of the prompt design. Therefore, we

[1] https://platform.openai.com/docs/guides/prompt-engineering
[2] https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/advanced-prompt-engineering
[3] https://microsoft.github.io/prompt-engineering/



Constraint	**Respond with a Python function in one code block.**
Persona	**As a software developer who follows best coding practices for maintainability such as avoiding code smells and writing simple and clean code,**
Chain-of-Thought	**Think carefully and logically, explaining your answer step by step.**
	Convert nanoseconds to a time in fixed format.
Few-shot examples	**For example, if the input is 4523 and 3600, the output is 01:15:23+01:00, and if the input is 4523605 and None then the output is "01:15:23".**
Signature	**The function signature is: def hydrate_time(nanoseconds, tz=None)**
Packages	**The function has access (but does not necessarily use) the following packages: time pytz datetime.**

Fig. 2. Example prompt in CodePromptEval.

follow an adapted pattern of few-shot prompting for code generation tasks also used in previous work [28], [33], where each example consists of a possible input of the function and its corresponding output. We use two input-output examples explained in natural language. We do not consider a varying number of shots.

- *Chain-of-Thought* (CoT) allows the LLM to break down the prompt by asking it to "think" step by step before solving the problem. This technique is used to prompt the LLM to perform explicit reasoning [34]. We apply Zero-shot-CoT [26] to isolate the impact of CoT from the few-shot prompt technique. In Zero-shot-CoT the steps that the LLM can follow are not explicitly mentioned in the prompt; rather, the model is expected to generate and follow its own reasoning process autonomously.
- *Persona* allows the LLM to play a specific role and consider its perspective when solving a problem [35], [36]. For the persona, we use the role of a software developer who focuses on practices and standards that software developers follow.
- *Signature* is a line of code that includes the signature of the function to generate. The signature includes the function name, the input parameters, and (optionally) the output.
- *Packages* is a list of libraries and files that exist in the environment in which the code runs. This includes local packages and external libraries. The packages used by a function are extracted by parsing all import statements in the Python file to collect the names of the modules and external libraries on which the function depends.

Previous work indicates that Signature and Packages are not necessarily used when prompting LLMs for code generation by developers [3]. While Signature is often a part of the prompt in code completion benchmarks [37], most code generation benchmarks construct prompts based on documentation (e.g., docstrings) without the signature [10]. Therefore, in this study, we present them as prompt techniques for code generation tasks as they might provide additional context that can guide the model toward more relevant code generation (e.g., types inferred by the parameter names).

B. CodePromptEval

To evaluate the different combinations of prompt techniques, we construct CodePromptEval – a dataset that includes 221 function-level code generation tasks, where each generation task is implemented using 32 different prompt variations. Each of these 32 prompts applies a unique combination of prompt techniques, resulting in a total of 7072 prompts (221 tasks × 32 variations).

To create our dataset, we initially start with the CoderEval Python dataset [10]. This dataset consists of 230 datapoints from 43 Python projects. Each datapoint consists of a prompt, a Python function (human-written baseline), and the corresponding tests (in form of unit tests or a main class). We first set up different virtual environments for functions from different projects, then we test the functions using the provided tests, and eliminate nine datapoints where the baseline does not pass the tests. This resulted in 221 datapoints that will be the foundation for our own CodePromptEval dataset.

Then, we ensure that the prompts are "pure" from any prompt technique that may be implicitly applied (e.g., providing examples), by going through the prompts manually and removing any elements that do not describe the purpose of the code. We then treat this prompt as a zero-shot prompt.

The next step was to prepare prompt templates by defining how each prompt technique will be implemented and mapping relevant information to prompt techniques. In particular, for each datapoint, we extract the signature of the function and the list of used packages (represented as imports at the beginning of the class). For chain-of-thought, we adapted the template recommended by Zhuosheng et al. [26]. To construct the persona, we defined a persona description of a software developer who follows best coding practices for maintainability. To implement the few-shot prompt technique, the first three authors of this paper manually constructed two input-output examples for each prompt following the template "If the input is X, then the output is Y". We also create corresponding tests to ensure that the input and output are correct. The examples were created based on the goal of covering both a typical (mainline) case and an edge case to ensure that the examples capture a range of expected behavior.

Finally, we define 32 prompt variations that we list in Table I. Each variation represents a prompt that applies a unique combination of prompt techniques. For example, **P7** is a prompt that provides the code signature, but uses no other prompt programming technique, whereas **P28** combines few-shot learning with CoT and the usage of the persona "software developer". **P8** is the zero-shot baseline, where no prompt technique is used and the model is only provided with the programming task. **P25** is the case where *all* prompt techniques are used in conjunction.

We then map each variation from Table I to the relevant information and templates for prompt techniques (e.g., imported libraries for packages), then we combine them with the 221 prompts from CoderEval, leading to CodePromptEval with 7072 concrete prompts (221 prompts times 32 variations) and their corresponding Python functions as ground truth.

The functions include both domain-specific implementations and commonly used utility logic extracted from GitHub repositories, rather than standard textbook algorithms such as sorting or searching. Table II describes the functions' length and cyclomatic complexity. Moreover, the 221 functions fall into six code

TABLE I
THE 32 COMBINATIONS OF PROMPT TECHNIQUES THAT WE CONSIDER IN OUR FULL FACTORIAL EXPERIMENT

ID	Few-Shot	CoT	Persona	Packages	Signature
P1	-	-	✓	✓	✓
P2	-	-	✓	✓	-
P3	-	-	✓	-	✓
P4	-	-	✓	-	-
P5	-	-	-	✓	✓
P6	-	-	-	✓	-
P7	-	-	-	-	✓
P8	-	-	-	-	-
P9	-	✓	✓	✓	✓
P10	-	✓	✓	✓	-
P11	-	✓	✓	-	✓
P12	-	✓	✓	-	-
P13	-	✓	-	✓	✓
P14	-	✓	-	✓	-
P15	-	✓	-	-	✓
P16	-	✓	-	-	-
P17	✓	-	✓	✓	✓
P18	✓	-	✓	✓	-
P19	✓	-	✓	-	✓
P20	✓	-	✓	-	-
P21	✓	-	-	✓	✓
P22	✓	-	-	✓	-
P23	✓	-	-	-	✓
P24	✓	-	-	-	-
P25	✓	✓	✓	✓	✓
P26	✓	✓	✓	✓	-
P27	✓	✓	✓	-	✓
P28	✓	✓	✓	-	-
P29	✓	✓	-	✓	✓
P30	✓	✓	-	✓	-
P31	✓	✓	-	-	✓
P32	✓	✓	-	-	-

TABLE II
FUNCTION STATISTICS IN CODEPROMPTEVAL

Metric	Min	Max	Mean	Std Dev
Number of Variables	0	32	2.52	4.35
Number of Parameters	0	7	1.70	1.30
Lines of Code	3	564	32.14	55.82
Block Depth	1	9	2.58	1.63
Cyclomatic Complexity	1	29	4.57	4.66

dependency levels: 33 self-contained (does not use packages outside the function scope), 25 standard library runnable (uses libraries available as part of Python standard library), 19 public library runnable (uses libraries available on PyPI), 54 class runnable (uses code outside the function, but within the class), 67 file runnable (uses code outside the class, but within the file), and 23 project runnable (uses code in other files). Our dataset, the virtual environments, and the few-shot examples and tests are provided in our replication package [38].

We illustrate an example prompt with all prompting techniques (**P25**) in Fig. 2. The prompt description, signature, and packages are extracted from CoderEval, while we construct the few-shot examples, persona, and chain-of-thought texts as a part of CodePromptEval. We also append a constraint at the beginning of each prompt to ensure that the output has a block of Python code with a self-contained function.

If different prompt techniques are combined, we apply them in a fixed order (as given in Fig. 2). This order ensures the sentence flows naturally. For example, the common practice is to place the persona at the beginning, and the few-shot examples after the purpose of the code. While it is possible to experiment with different orders of prompt techniques in a prompt, we consider this outside the scope of this study.

C. Code Generation

We focus on LLMs with a decoder-only transformer architecture, which is at the time of writing the preferred architecture to use in code generation tasks [39]. Therefore, we select the following LLMs for our study: GPT-4o, Llama3-70B-Instruct, and Mistral-Small-Instruct-2409 (22B). We also collect data for two previous-generation LLMs (GPT-3.5-turbo and Llama2-7B-Instruct), but omit discussing the results for these older models for reasons of brevity in this paper. In general, the results for these older models showed lower passing rates and higher complexities. However, the overall impact of the techniques remained consistent with the findings we report for the studied LLMs, albeit with varying significance levels. The collected data for these models is still available in our replication package [38].

We run all 7072 prompts on the selected LLMs three times to account for the non-deterministic nature of LLMs. For all LLMs, we set the temperature to 0.2, which has been commonly used for code generation tasks [28], [40], and complies with the recommendations for our correctness measure [28]. For the API-based GPT models, we send requests to the external API and store the responses. We host the remaining models on the Alvis cluster, a NAISS resource (National Academic Infrastructure for Supercomputing in Sweden) dedicated to Artificial Intelligence and Machine Learning research[4] using models downloaded from Huggingface[5]. Running the self-hosted LLMs on Alvis required around 2800 GPU hours using Nvidia A100 GPUs. For the GPT models, we use the OpenAI API, which is billed based on the tokens that are processed. To run GPT-4o and GPT-3-turbo on all the prompts in CodePromptEval, we provide around 2.45 million input tokens and generate approximately 7.66 million output tokens.

D. Evaluating the LLM-Generated Functions

After generating 7072 code solutions three times, we evaluate them based on three main aspects following our research questions (correctness, similarity, and quality). We use different tests to measure statistically significant differences for the measures below, hence we detail the choice of statistical methods in their corresponding results sections.

Correctness: To evaluate their correctness, we run the generated functions against their corresponding tests in CoderEval. When running the functions, we replace the generated function name with the originally expected one to ensure compliance with our test cases.

There are two types of tests in CoderEval: Python unit tests, and a main function with different statements and conditions that set a boolean variable `isT` (is True) to `False` when at least one of the conditions does not hold. To ensure consistency

[4] https://www.c3se.chalmers.se/about/Alvis/
[5] https://huggingface.co

and instrumentation of our experiment, ensure that an AssertionError is thrown when needed, by adding an assert statement at the end of tests in the form of a main function `assert isT`. Furthermore, as some of the LLM-generated functions can be erroneous and get stuck in an infinite loop, we wrap the tests with error-handling constructs (a `try/except`) and set a timeout of 60 seconds per function. Then we collect the test results and the error messages when applicable.

We distinguish syntactic correctness and semantic correctness of the function. The Python function is syntactically correct if its syntax is valid and the function is runnable. Semantically correct functions are functions that pass their corresponding tests. When a function is both syntactically and semantically correct, then it is labeled as plausibly correct [41]. In the remainder of the paper, we use correctness as a short-hand for plausibly correct.

Similarity: To assess the LLM-generated code's similarity to the ground truth obtained from CoderEval (human-written functions), we measure the CrystalBLEU score [42]. CrystalBLEU combines four n-gram measures where $n = 1, 2, 3, 4$ while accounting for "trivial grams" that are shared across all functions, such as Python keywords. The combined n-grams that are used as a metric for syntactic similarity are then used as a proxy to estimate the semantic similarity.

Quality: Regarding code quality, we focus on measures that are related to maintainability [43] and we only measure them for functions that pass their tests. In other words, we measure the quality only for functionally correct functions. We use Pylint[6] to generate a report with identified code smells in the generated functions. Moreover, we compute the code complexity for both the LLM-generated functions and the equivalent ground truth (i.e., the human-written functions in CoderEval) to compare both results and see how the different prompts have an impact on the code quality. Code complexity refers to how detailed and interconnected different parts of the code are, which can make the code harder to understand and test. To get an overview of the complexity of the generated functions, we measure McCabe's cyclomatic complexity via the Radon Python package[7] and cognitive complexity [44] via the cognitive-complexity Python package.[8]

IV. CodePromptEval Overview

In this section, we provide an overview of the aggregate results from running three LLMs (GPT-4o, Llama-3, and Mistral) three times on the CodePromptEval dataset. This section answers RQ1 in our study.

Note that these results are not an assessment of the capabilities of these models when used with an "ideal" prompt, but an aggregation over all prompt technique combinations in our study. That is, the following results should be read as an overview of CodePromptEval, and not as a judgment of which LLM performs best in general. Detailed drill-downs assessing the performance of individual (combinations of) prompt techniques will be presented in Section V.

TABLE III
Overview of Passed and Failed Functions per LLM. We Also Show the Breakdown of Failures Types (Total = 7072 Functions, Averaged Over 3 Runs)

Results	GPT-4o	Llama3-70B	Mistral-22B
Passed	3691 ± 12 (52.2%)	3575 ± 11 (50.5%)	3318 ± 13 (46.9%)
Failed	3381 ± 12 (47.8%)	3497 ± 11 (49.5%)	3754 ± 13 (53.1%)
- Syntactic	27 ± 1 (0.4%)	59 ± 4 (0.9%)	103 ± 3 (1.5%)
- Semantic	1303 ± 23 (18.4%)	1415 ± 15 (20.0%)	1182 ± 18 (16.7%)
- Operational	2051 ± 21 (29.0%)	2023 ± 6 (28.6%)	2468 ± 16 (34.9%)

A high-level results summary is shown in Table III. There are a total of 7072 generation tasks in the dataset. All three models are able to solve (generate functions that pass all tests) approximately half of the tasks. Mistral performs worst in our study, solving on average 3318 (46.9%) of tasks, and GPT-4o does best solving 3691 (52.2%), outperforming the worst model by approximately 5 percentage points.

To get a better idea of whether these results are impacted by the code level of the function, we use the code levels defined by Yu et al. [10] that are based on the nature of dependencies of the function. The code levels are: self-contained, standard library runnable, public library runnable, class runnable, file runnable, and project runnable. Code levels provide a rough indication of the "difficulty" of a generation task, based on what kind of dependencies the LLM needs to correctly incorporate.

We looked into the code levels that passing and failing functions belong to (see Fig. 3). Unsurprisingly, the fail rate for all models increases as tasks get more difficult (i.e., by construction, class runnable tasks tend to be substantially more challenging than self-contained ones, and all models struggle much more with solving them correctly). Pass rates for the easiest type of task (standard library runnable) are close to 90% for all models, going down to as low as 31% to 41% for the most challenging tasks (class runnable). We observe that, overall, all three models perform comparably on most code levels, with the exception of self-contained tests (where GPT-4o outperforms the other models by a larger margin of 10 to 13 percentage points). This difference explains most of the slightly higher overall performance of GPT-4o. We also confirmed these differences using Chi-square test, which assesses the association between categorical variables (code level and pass/fail outcome) resulting in p-value < 0.0001, and Cramér's ϕ as an effect size for the relationship between the two nominal variables ($\phi = 0.34$:- medium effect).

Finally, we report what errors led to the failing tests shown in Table III and Fig. 3. We report the error types based on the Python exception that is first thrown when running the tests. The results of the error types are visualized in Fig. 4. The most common error type for failed tests across the LLMs is `AssertionError`, indicating that the LLM generated a Python function that did not exhibit precisely the expected functionality (as defined through unseen tests). However, are also frequently encountered such as `TypeErrors` (operation

[6] https://pypi.org/project/pylint/
[7] https://pypi.org/project/radon/
[8] https://pypi.org/project/cognitive-complexity/

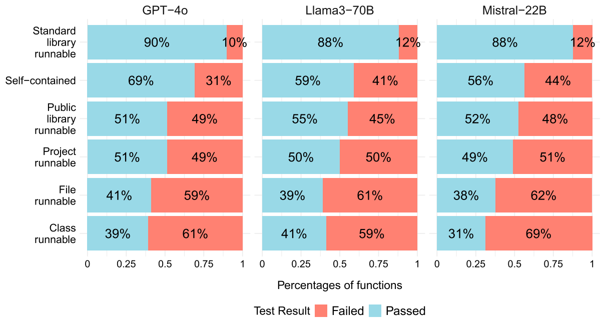


Fig. 3. Passed and failed functions per LLM for each code level across three runs. The total number of functions per LLM in a single run is 7072.

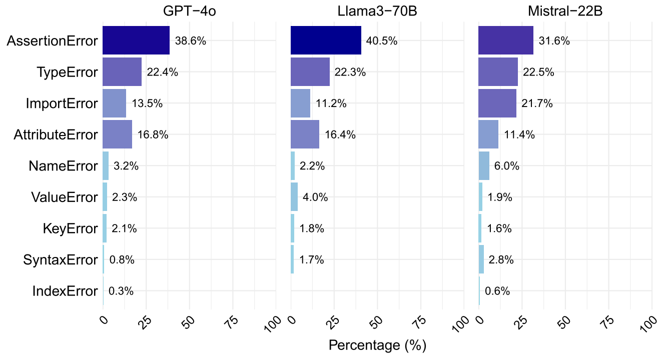


Fig. 4. Percentages of error types occurring among failing tests for functions generated by GPT-4o, Llama3-70B, and mistral-22B across three runs.

is performed on a value of an inappropriate type, indicating that the LLM misjudged the runtime type of a Python object), `AttributeErrors` (invalid attribute reference is made), and `ImportErrors` (a faulty import of a module or object). Other errors, such as `NameErrors` or `IndexErrors`, exist but are rare. While there are differences between the LLMs, they are relatively minor and not systematic. The most notable difference is that Mistral tends to generate functions leading to an `ImportError` or `NameError` more frequently than the other LLMs, whereas `AttributeErrors` are less frequent in Mistral-generated code.

Key Findings (RQ1): Overall, we observed that GPT-4o minorly outperforms the other LLMs in the study. However, in general, results are consistent between current-generation LLMs. Depending on task difficulty, all LLMs can solve between 31% and 90% of tasks. Assertion and TypeErrors are the most common cause of failed tests.

V. Prompt Technique Comparison

We now turn towards RQ2, and describe the results of a statistical analysis examining how the different prompt techniques applied in each prompt impact the function regarding (i) correctness, (ii) similarity to the ground truth, and (iii) quality.

A. *Correctness*

A central question for assessing the value of prompt techniques is how likely a (combination of) techniques is to lead to a correct code, meaning that it is both (i) syntactically and operationally correct (valid and does not throw errors) and (ii) semantically correct (passes the tests). To measure the correctness of different prompts, we use the well-established Pass@k metric [28]. This metric measures the likelihood of drawing k passing functions from the results of n number of generations (or repetitions).

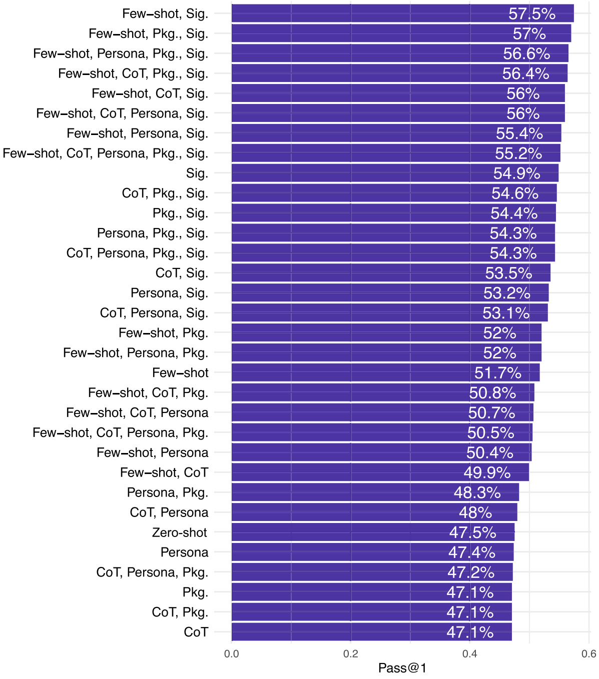


Fig. 5. Pass@1 results of the different (combinations of) prompt techniques exemplified for GPT-4o.

In our study, we run the functions generated by the 32 combinations of prompt techniques over three repetitions ($n = 3$) on the tests provided by the CoderEval benchmark. Then, we collect the test results (pass or fail) and measure Pass@1 ($k = 1$) accordingly. Fig. 5 shows Pass@1 results for all combinations of techniques (see Table I) for GPT-4o. Given that results between different models appear to be very consistent (see also Section V), we focus our discussion on one example model. However, results for the other models can be found in the supplemental material.

It is evident from Fig. 5 that the most important technique when it comes to correctness is the presence of a function signature. Combining the signature with other techniques, such as few-shot or chain-of-thought, is sometimes helpful to further increase the likelihood of a generated function being correct (albeit by a very small margin, e.g., adding chain-of-thought and few-shot examples to the signature only leads to an improvement of 0.1 percentage points). The best combination, with a Pass@1 of 57.5%, is the combination of signature and few-shot. We achieved the worst results in terms of correctness when using chain-of-thought alone, with a Pass@1 of 47.1%. It is surprising to note that the impact of prompt engineering techniques is overall lower than we would have expected — the difference between the best and worst combinations is merely

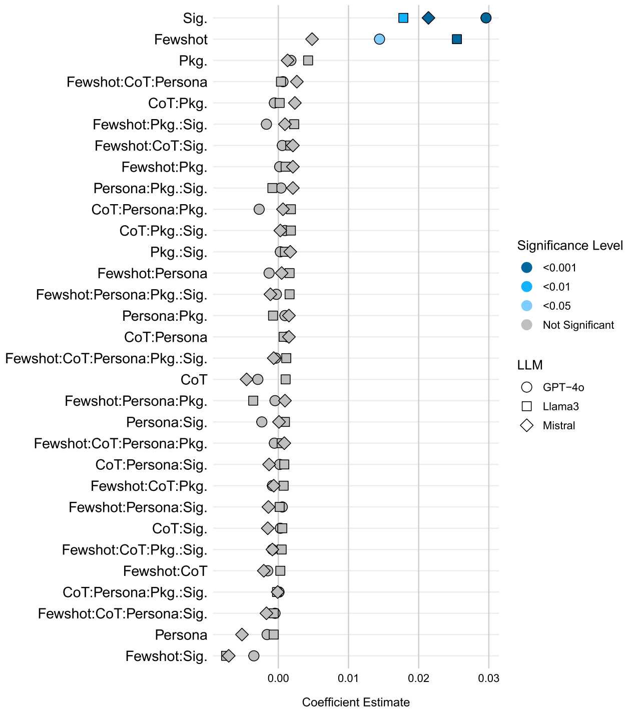


Fig. 6. Results from our regression analysis for the pass@1 scores. Each point visualizes the coefficient estimate for the corresponding combination. The darker colors represent more conservative significance levels (α). zero-shot is not depicted, as it cannot be combined with other techniques.

10 percentage points, i.e., prompt programming seems to have a noticeable impact in only a little over one in ten generation tasks.

Our findings also indicate that sometimes the addition of more information in the prompt leads to worse performance. For example, using only few-shot and signature performs better than if all possible prompt techniques are used. Further, it is evident that techniques can interact in non-obvious ways. For example, both package information and CoT alone led to the worst Pass@1 results. However, if these techniques are used in conjunction with a function signature, Pass@1 improves marginally over using only the signature in isolation.

To further investigate these interactions between factors in our experiment, we conducted a multi-linear regression analysis. Fig. 6 shows the five prompt techniques in the study their interactions and their effect on the pass@1 score. For instance, “CoT:Persona” describes if the impact on test results comes from the interaction of CoT and Persona in a prompt, regardless of whether that prompt includes other prompt techniques. Similarly, “Sig.” (signature) refers to all prompts that include at least the signature (including, for example, P23, the combination of few-shot and signature), and is not limited to prompts that only specify the signature.

The multi-linear regression results in a coefficient estimate and a p-value for each factor and possible interactions among the factors. The coefficient reflects the impact on the test results, positive and negative coefficients refer to positive and negative impacts, respectively. The p-value indicates how significant the impact is.

In line with our previous findings, we observe that the presence of a signature and few-shot in a prompt (regardless of whether they are combined with other prompt techniques) affect the test results positively (albeit with different statistical significance levels for different LLMs), and a positive, high coefficient estimates (meaning there is a significant positive impact on correctness). Interestingly, few-shot does not have a statistically significant impact in the case of the Mistral model.

The remaining main factors (packages, chain-of-thought, and persona) do not have a statistically significant impact on any of the three models.

> **Key Findings (RQ2.1):** The presence of a signature or few-shot has the clearest positive impact on correctness. The other prompt techniques in the study do not have a statistically significant impact on correctness. However, in general, the difference between “good” and “bad” prompt techniques is surprisingly low. Adding additional information to a prompt sometimes leads to worse performance.

Digging deeper into what causes generated functions to fail, Fig. 7 displays the percentages of errors encountered for each combination of prompt techniques. We show the four most common error types (`AssertionError`, `TypeError`, `AttributeError`, and `ImportError`) using GPT-4o (other models in the supplemental material). For example, 48.1% of the failed functions of prompts with few-shot, packages, and signature throw an `AssertionError`.

The combinations of prompt techniques are ordered from the fewest errors (at the top) to the most errors (at the bottom). In general, we see that the prompts that result in the least number of errors (among the first rows in the heatmap) are combinations that include a signature. On the other end, the prompts with the most errors lack few-shot examples.

Taking a closer look at the different error types, we observe that while `AssertionErrors` generally occur at a higher rate than the other error types across all prompt techniques, they are particularly more frequent in prompts that include the function signature, meaning that failed functions by prompts with a function signature are able to run but fail their tests due to a semantics-related error. In contrast, the absence of the function signature often leads to `ImportErrors` as well as `TypeErrors` that primarily occur because the LLM misjudges the expected number or order of positional arguments when generating functions.

Interestingly, in a subset of cases, `ImportErrors` occurred even when packages were explicitly specified. To investigate this, we manually inspected five random prompts where packages were specified but still resulted in `ImportErrors`. We found that when the prompt indicated the use of a package that is local or unfamiliar to the LLM, the LLM hallucinated and attempted to import non-existent functions from the specified packages.

We note that these findings are consistent for GPT-4o and Llama3, while the errors of code generated by Mistral lacked any clear patterns for the above mentioned error types. However, we observed a trend of a higher rate of `AttributeErrors` in Mistral when the signature is included in the prompt.

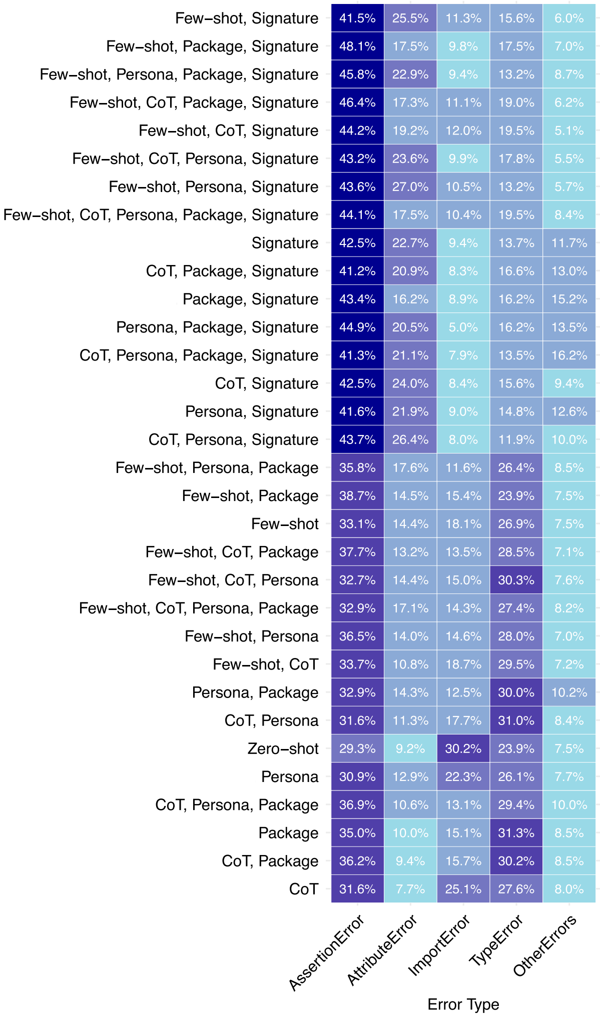


Fig. 7. The percentages of error types that we observed in failed functions generated by different combinations of prompt techniques (GPT-4o) over three runs.

For the other two LLMs (GPT-4o and Llama3), we did not observe any consistent patterns among the prompts that triggered `AttributeErrors`.

Overall, we emphasize that encountering a certain error does not necessarily mean that the function is free from the other error types, as the program terminates at the first error thrown. However, assertions are evaluated after the function has successfully been executed, so an `AssertionError` indeed indicates that no other errors have occurred. Further, `AssertionErrors` are qualitatively different from other error types, as they do not indicate a fundamentally broken function, but rather that the LLM misunderstood (or could not correctly guess from context) some assumptions about the functionality of the code that is to be generated.

Key Findings (RQ2.1): Including the signature or few-shot examples in prompts generally reduces errors, particularly TypeErrors, AttributeErrors, and ImportErrors. Providing package information can naturally reduce ImportErrors but may cause hallucinated imports if unfamiliar to the LLM.

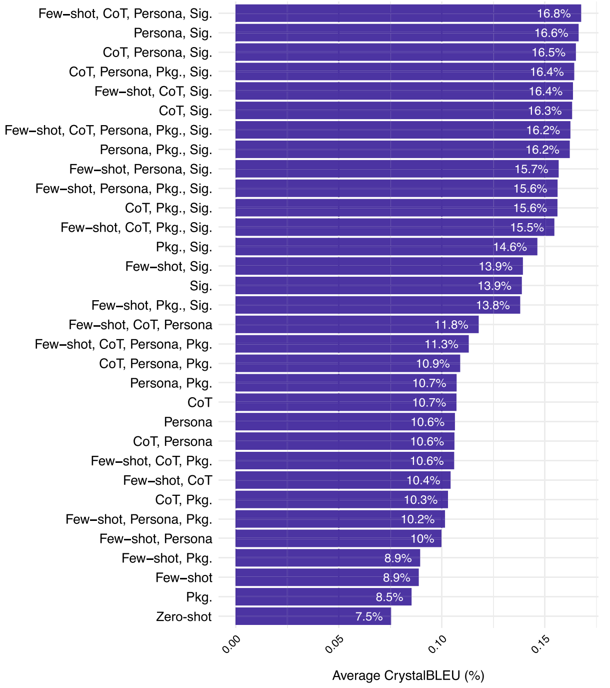


Fig. 8. The average CrystalBLEU scores for the functions generated by each combination (Llama3).

B. Similarity

Beyond correctness, we believe that another important question is how similar generated functions are to the human-written baseline. We use the CrystalBLEU score [42] to measure how similar the generated function is to the baseline in terms of the syntax and semantics of the function combined. CrystalBLEU is seen as a stricter improvement over the older CodeBLEU metric [29]. In our analysis, we remove the signature of the generated function and the ground truth before measuring the similarity to avoid any bias toward the signature prompt technique.

In Fig. 8, we see that the average CrystalBLEU score across three runs is low for all approaches (varying between 16.8% and 7.5% for Llama3) indicating that generated solutions are largely different than how humans have solved the same tasks. Results for the other models are in the supplemental material.

We observe that using any prompt technique increases similarity (i.e., zero-shot has the lowest similarity to the baseline for all three models). Consistently with correctness, combinations that include a signature lead to higher similarity. This is unsurprising, given that a predefined signature restricts the solution space for the LLM (which can be desirable or unwanted depending on context). Combining more techniques indeed seems to generally increase similarity. We also observe that few-shot can decrease the similarity, achieving a score of 8.5%. However, combining it with signature, chain-of-thought or persona can improve the similarity to 10% and above.

To better understand the impact of the prompt techniques and their interactions on the code similarity, we now perform a linear mixed-effects regression analysis to see how the different prompt techniques and their interactions can impact the

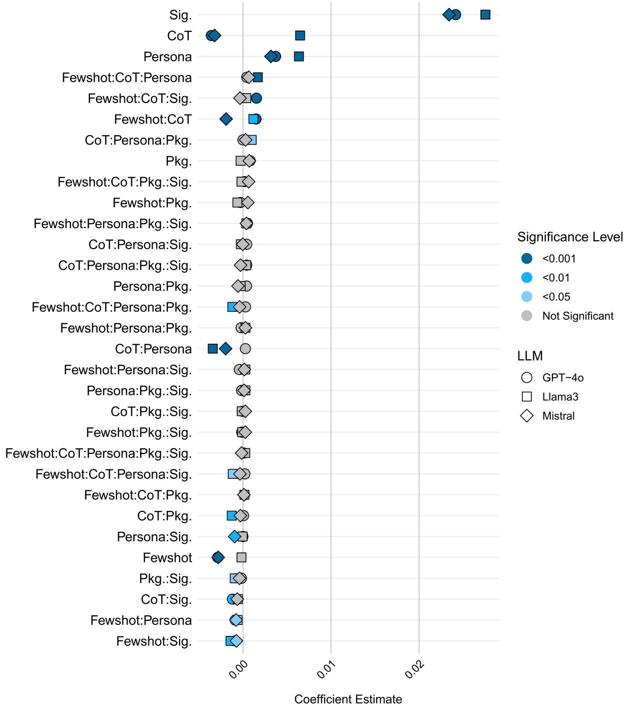


Fig. 9. The coefficient estimates from the linear mixed-effects regression of prompt technique combinations that significantly impact the CrystalBLEU score.

CrystalBLEU while accounting for the within-group variation (random effects) that arise from the three runs of each LLM.

Fig. 9 shows our linear mixed-effects regression results. We see that, regardless of the test results, the presence of a signature or persona in a prompt can significantly increase the CrystalBLEU score. Chain-of-thought (CoT) seems to also positively impact the CrystalBLEU score for Llama3, but significantly lower it for GPT-4o and Mistral. We also observed that the interaction between certain prompt techniques can either reinforce or counteract the effects seen when the techniques are used individually. For example, while both Signature and CoT independently increase similarity in Llama-generated functions, using them together in a prompt can reverse that positive effect and significantly reduce similarity.

Key Findings (RQ2.2): The signature and persona increase the overall similarity of the function to the baseline (i.e., code written by humans). Few-shot decreases the similarity unless combined with chain-of-thought.

C. Quality

Using prompt techniques that yield correct functions does not necessarily mean that these functions are maintainable and of good quality. Hence, we now turn to an assessment of the quality of the generated code. In our experiment, we focus on code smells and complexity as proxies of code quality. For this analysis, we only evaluate functions that *pass* their tests (see Section V-A). We do not believe that assessing the quality of functionally incorrect implementations is fruitful because refactoring must be done on a working piece of code and preserve its behavior [45].

TABLE IV
List of Code Smells in Generated Functions

Category	Code Smell ID	Definition
Error	E0602	Usage of an undefined variable.
Warning	W0611	Import statement not used.
Warning	W0613	Function argument is not used.
Refactoring	R0903	Insufficient public methods in a class.
Refactoring	R1705	Unnecessary “else” after “return”.
Convention	C0301	Line exceeds the character limit.
Convention	C0103	Violating UPPER_CASE naming style.
Convention	C0115	Class lacks a descriptive docstring.
Convention	C0116	Function lacks a descriptive docstring.
Convention	C0411	Wrong import order.
Convention	C0304	File missing a final newline.

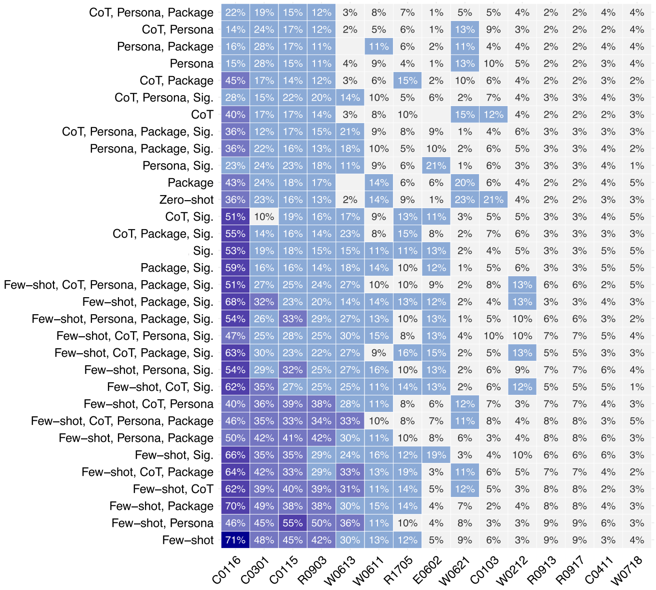


Fig. 10. Percentages of functions generated by GPT-4o that have different code smells. Empty fields indicate that no smell of this type is found. Note that functions can have instances of multiple types of smells.

For code smells, we run Pylint on the generated functions for each prompt in CodePromptEval, using the code smell IDs defined by Pylint. Then, we group the code smells for prompts that share the same prompt techniques, and finally, we select the top 15 code smells that were the most frequent across all prompt techniques.

We find 11 code smells that fulfill these criteria for all LLMs (see Table IV). Most identified code smells are *convention* code smells, but there is also one *error*, two warnings, and two *refactoring* smells. From this list, we decided to remove C0304 as it is present in all generated functions across all LLMs and is mostly an artifact of our generation pipeline.

In Fig. 10, we show what percentage of functions have at least one instance of each code smell. For reasons of brevity, we focus on GPT-4o (Llama3 and Mistral's in the supplemental material).

We note that 71% of the functions generated using the few-shot technique contain C0116 code smells, indicating that these functions lacked a descriptive docstring (in contrast to only 22%

TABLE V

CYCLOMATIC COMPLEXITY ANALYSIS USING WILCOXON TEST AND A12 VARGHA DELANEY FOR THE EFFECT SIZE (N- NEGLIGIBLE, S- SMALL, M- MEDIUM, L- LARGE). ↓ INDICATES A REDUCTION IN COMPLEXITY, ∅ INDICATES NO STATISTICAL DIFFERENCE

Combinations	**GPT-4o**			**Llama3**			**Mistral**		
	p-value	**A12**	**Effect**	**p-value**	**A12**	**Effect**	**p-value**	**A12**	**Effect**
Package	**0.0015**	**0.403**	**↓ (S)**	**0.0017**	**0.410**	**↓ (S)**	**0.0050**	**0.412**	**↓ (S)**
Persona, Package	**0.0136**	**0.442**	**↓ (S)**	**0.0033**	**0.412**	**↓ (S)**	**0.0046**	**0.424**	**↓ (S)**
Zero-shot	**0.0172**	**0.445**	**↓ (S)**	**0.0009**	**0.401**	**↓ (S)**	**0.0001**	**0.384**	**↓ (S)**
CoT, Package	**0.0188**	**0.448**	**↓ (S)**	**0.0118**	**0.442**	**↓ (S)**	**0.0016**	**0.409**	**↓ (S)**
CoT, Persona, Package	0.0213	0.451	↓ (N)	**0.0015**	**0.431**	**↓ (S)**	**0.0067**	**0.433**	**↓ (S)**
Persona, Sig.	0.0755	0.475	∅	**0.0230**	**0.440**	**↓ (S)**	**0.0041**	**0.442**	**↓ (S)**
Persona	0.0524	0.466	∅	**0.0015**	**0.401**	**↓ (S)**	**0.0004**	**0.408**	**↓ (S)**
Sig.	0.0909	0.460	∅	**0.0093**	**0.430**	**↓ (S)**	**0.0076**	**0.444**	**↓ (S)**
Package, Signature	0.0295	0.460	↓ (N)	**0.0398**	**0.436**	**↓ (S)**	0.0143	0.458	↓ (N)

of functions generated by chain-of-thought combined with a persona and package information). In general, we observe that prompts that apply the few-shot and signature prompt techniques generate functions with more code smells and, more specifically, warning and error code smells compared to other prompts.

On the other hand, we observed that CoT, persona, and package lead to functions with fewer code smells, unless these prompt techniques are combined with few-shot and/or signature, then the percentage of code smells increases. This is interesting, as we have seen that few-shot and signature are the techniques with the clearest positive impact on correctness (see Section V-A). In part, this discrepancy could be explained by solutions for challenging tasks that LLMs only solve correctly when provided examples or a signature (recall that, for this analysis, we have only investigated functions that pass all tests — hence, some challenging functions have an analyzable solution for signature and few-shot, but not other techniques). However, we note that the differences in Fig. 10 are too large to be entirely explained in this way. Consequently, we conclude that CoT, persona, and package information indeed seem to systematically lead to fewer code smells.

Key Finding (RQ2.3): While using CoT, persona, or package information leads to fewer correct solutions, these techniques lead to higher-quality code in terms of code smells.

We now turn towards the cyclomatic and cognitive complexity and compare the complexity of generated solutions to the complexity of the human-written baseline. In Table V, we show the p-values resulting from the *paired* Wilcoxon test to assess the statistical significance of differences between the cyclomatic complexity of the generated functions and ground truth. We only show the prompt techniques that had a significant impact on the complexity for at least two LLMs ($\alpha = 0.05$). Complete results are in the supplemental material.

We use Vargha Delaney A12 measure [46] to understand the nature of the impact (reduces or increases complexity) and to quantify the effect size (Negligible ($A_{12} \geq 0.45$), Small ($0.36 \leq A_{12} < 0.45$), Medium ($0.29 \leq A_{12} < 0.36$), or Large ($A_{12} \leq 0.29$)). Vargha Delaney A12 is a probability measure (that was later adopted as an effect size measure), which describes the probability that one level (generated function complexity) is greater than a corresponding value in another level (ground truth complexity). If the A12 is less than 0.50, it means that the values of the first level are lower than the second level, and the lower the score is, the larger the effect size. This allows us to see if the prompts generate functions with a significantly lower or higher complexity as the ground truth, or with a comparable complexity when no significance is observed.

Similar to the code smells results, we see that CoT, persona, and packages reduce the complexity in comparison to the baseline. A zero-shot prompt also leads to lowered complexity. However, all reductions have (at most) a small effect size. This can be explained by the low cyclomatic complexity of all LLM tasks — in general, only minor simplifications are even possible to the generally rather simple code snippets.

For cognitive complexity (see Table VI), we observe larger differences among the LLMs than between combinations of prompt techniques in general. There was no combination of prompt techniques that reduced the cognitive complexity across all three LLMs. GPT-4o seems to generate functions with no or small differences to the ground truth. Mistral can reduce the cognitive complexity with a small effect size when the prompt does not include few-shot and a persona, packages or CoT applied in the prompt. In contrast, there are no clear trends or patterns among the prompts in Llama3 rather most of the prompt techniques seem to reduce the cognitive complexity with a small effect size. We conclude that Llama3 appears to lead to simpler solutions than the other models, particularly GPT-4o.

It is interesting to observe that no combination of prompt techniques leads to *more complex* solutions than the baseline — generated solutions are always slightly simpler or comparably complex. Viewed positively, this may indicate that LLMs generate rather clean code. However, a more negative interpretation may also be that the generated code does not cover some complex corner cases that human-written solutions account for (which may not be covered by CoderEval tests).

Key Findings (RQ2.3): There are noticeable differences among models with regard to the complexity of the code they produce. Llama3 appears to produce simpler solutions systematically. There were no cases of increased complexity — LLM solutions were comparably complex to human-written code, or simpler.

TABLE VI
COGNITIVE COMPLEXITY ANALYSIS USING WILCOXON TEST AND A12 VARGHA DELANEY FOR THE EFFECT SIZE (N- NEGLIGIBLE, S- SMALL, M- MEDIUM, L- LARGE). ↓ INDICATES A REDUCTION IN COMPLEXITY, ∅ INDICATES NO STATISTICAL DIFFERENCE

Combinations	**GPT-4o**			**Llama3**			**Mistral**		
	p-value	**A12**	**Effect**	**p-value**	**A12**	**Effect**	**p-value**	**A12**	**Effect**
Few-shot, Persona	**0.0239**	**0.437**	**↓ (S)**	**0.0106**	**0.403**	**↓ (S)**	0.2629	0.461	∅
Few-shot, Persona, Package	**0.0326**	**0.445**	**↓ (S)**	**0.0210**	**0.419**	**↓ (S)**	0.8239	0.505	∅
Persona, Sig.	0.1846	0.473	∅	**0.0024**	**0.417**	**↓ (S)**	0.0438	0.465	↓ (N)
CoT, Persona	0.4080	0.524	∅	**0.0212**	**0.441**	**↓ (S)**	**0.0052**	**0.445**	**↓ (S)**
CoT, Package	0.1694	0.485	∅	**0.0245**	**0.448**	**↓ (S)**	**0.0334**	**0.435**	**↓ (S)**
CoT	0.3593	0.516	∅	**0.0444**	**0.458**	**↓ (S)**	**0.0093**	**0.432**	**↓ (S)**
Persona	0.2361	0.513	∅	**0.0027**	**0.398**	**↓ (S)**	**0.0155**	**0.442**	**↓ (S)**
Package, Sig.	0.1617	0.468	∅	**0.0003**	**0.406**	**↓ (S)**	**0.0437**	**0.448**	**↓ (S)**
Package	0.1694	0.485	∅	**0.0006**	**0.383**	**↓ (S)**	**0.0267**	**0.426**	**↓ (S)**
Zero-shot	0.2522	0.504	∅	**0.0008**	**0.385**	**↓ (S)**	**0.0298**	**0.415**	**↓ (S)**

VI. DISCUSSION

In this section, we discuss the key lessons learned from this study, the implications of our findings for software engineering practitioners and researchers, as well as validity threats.

A. Lessons Learned

L1: The differences in the results of prompt techniques are not dramatic: We carefully designed a full factorial experiment to evaluate not only prompt techniques but also combinations of them in a prompt. Our analyses revealed that, while there was an impact of some prompt techniques on the generated functions, the results for most of the prompt techniques were not that different. For example, the difference in the Pass@1 rates for the prompts with the highest and lowest rates is only around 10 percentage points (see Fig. 5), and the effect sizes of the complexities are mostly small or insignificant (see Table V). These insights align with other studies that evaluate prompt techniques on code summarization [18] and generation [22], where the performance results of different prompt techniques such as CoT, few-shot, self-collaboration, among others, are also comparable. In contrast, we see clearer differences in the performance results of some prompt techniques when using benchmarks for math-related tasks or general question-answering [26]. We conclude that a strong emphasis on prompt programming is not necessary in the context of function-level code generation using current-generation models.

L2: Providing information about the interface via few-shot or signature is useful, but limits the "creativity" of the LLM: In our correctness and similarity results, the signature and few-shot prompt techniques stood out among other prompt techniques. In general, we believe that while they are two different prompt techniques, they can provide similar context about the expected functional interface in terms of positional arguments and expected output. This was also revealed through our general multi-linear regression results in Figs. 6 and 9, where we see that having either signature or few-shot examples significantly impact the code's correctness or similarity, but their interaction or combination does not help. In relation to previous work by Ahmed et al. [8], [47], we observe a similar pattern where contextual information about the parameters and other identifiers can improve the code summarization. However, providing this information limits the solution space for the LLM (i.e., it restricts the potential for "creativity"), which may not always be desired.

L3: There is a trade-off between correctness and maintainability when choosing prompt techniques: Our analysis revealed contrasting results: prompts with few-shot examples or function signatures improved correctness but increased complexity and number of code smells, while prompts that employed persona, CoT or package had lower passing rates but significantly enhanced code maintainability (see Tables V, VI for complexity and Fig. 10 for code smells). While previous research suggests that the use of a persona in the prompt does not improve the outcome [48] but can improve the personalization and user experience [27], we believe that this only applies to simple personas such as "software developer". However, our results indicate that personas can be more beneficial when used as a way to induce additional quality requirements e.g., "software developer who writes clean and simple code". Recent work has also shown that personas can be beneficial for code generation when used in more complex approaches such as self-collaboration where multiple personas (e.g., requirement engineer, software tester, and a developer) are used together to iteratively construct the code in a systematic way [24].

B. Implications

I1: Researchers should prioritize refining prompts for more effective prompt programming experiments We shed light on two components of experiments in prompt programming: the generation tasks, and the prompts. Based on observations in our previous work [3], we note that developers often use LLMs for more complex tasks than those in common datasets such as HumanEval [37] or CoderEval [10]. Although we were able to analyze and compare different prompt techniques, we believe that a dataset with are more representative functions of the large systems and projects that developers typically work with is needed.

Further, prompts in common benchmarks often lack a consistent format or level of detail. For instance, the prompts in CoderEval are based on functions' docstrings rather than actual prompts. Sclar et al. [49] show that LLMs, regardless of their sizes and number of parameters, are highly sensitive to small prompt changes such as prompt formatting. We observed similar behavior when experimenting with the template *"The* *function uses the following packages"* for the packages prompt technique and found that it caused errors related to using the wrong packages. We traced the issue back to the prompt itself and realized that the packages listed were not necessarily used by the function but existed in its class. When we modified the template to *"The function has access to (but does not necessarily use) the following packages,"* we mitigated the issue. This pre-processing of the prompt technique templates is another aspect of prompt programming recommended by Obrien et al. [50]. We found value in inspecting and refining prompts and creating our own few-shot examples, which increased our confidence in the dataset's reliability and stability. Therefore, we encourage researchers to invest in similar efforts.

I2: Software developers should avoid overusing prompt techniques While we saw that prompt techniques can be beneficial for certain criteria (e.g., signature for correctness and persona for quality), we also saw that combining them does not necessarily yield better results. In fact, some cases showed that including an additional prompt technique can cancel out the impact of the existing prompt techniques. For instance, in the code smells results in Fig. 10, we show how the inclusion of few-shot examples to CoT and persona can increase the code smells by more than one-third. Previous work has also shown how few-shot examples can hurt the LLM performance if not carefully engineered by humans [26].

I3: Different LLMs have different sensitivity levels to the prompt techniques We argue that a single prompt technique does not have the same impact on the different aspects (correctness, similarity, or quality) of the generated code across all LLMs. We have seen that the three LLMs demonstrated different sensitivity levels to the prompt. For instance, we saw how the similarity scores of Llama3-generated functions were significantly impacted by CoT, while it showed no effect for GPT-4o and Mistral (see Fig. 9). The error types for Mistral did not seem to be strongly impacted by prompt techniques as they did for other LLMs. Note that these differences in the models do not necessarily come from the model size and the number of parameters it was trained on, but rather the underlying architecture it uses (which aligns with findings by Wang et al. [18]). This implies that when a software company integrates an LLM into its processes and provides employee training, it should develop specific guidelines tailored to the LLM, including recommendations for prompt techniques that align with the model's characteristics e.g., if an LLM returns simple functions in general, so prompt techniques that impact complexity may not be needed.

I4: Determining the purpose of the code generation is essential for the use of prompt programming Depending on whether the intended use of the LLM is to support human developers or to completely automate code generation, prompt programming has different significance. We manually inspected 40 randomly selected failing functions from different code levels, each of which had passed with at least one other prompt, to understand what caused the failures. We saw that while the use of some prompt techniques has significantly minimized the number of errors and code smells, many of these issues can be easily fixed by human developers, arguably requiring less time and effort than re-prompting the LLM and applying an additional prompt technique. For instance, the absence of a signature in a prompt causes `TypeErrors` when the LLM misjudges the number of arguments, or misses that the function is a part of a class until the signature with a `self` parameter is provided. Moreover, prompting the LLM with few-shot examples reduced `AssertionErrors` mostly because the original prompt lacked clear specifications for edge cases and input/output formats, which could be picked up from the examples and result in passed tests (see Fig. 7). On the other hand, prompt programming can be more valuable when the purpose is to automate code generation and return correct and maintainable code without the need for human intervention, especially to apply simple modifications or refactoring actions.

C. Threats to Validity

External validity. The main threats to external validity in this study are associated with the prompt techniques, the LLMs and the benchmark we utilized. There are many possible prompt techniques that can potentially impact code generation, such as self-collaboration [31], AceCoder [32], or providing the whole class as context. However, we decided to select common prompt techniques that can be practically applied by a typical software developer in most code generation tasks. Another important question is whether the use of more powerful LLMs can result in different findings and eliminate the need for prompt programming. We used three current-generation LLMs during the study, including GPT-4o (200B parameters) and Llama3 (70B parameters). Our replication package [38] also includes results for older LLMs (GPT-3.5, Llama2), showing that the prompt techniques affecting code generation in this study similarly impact older models.

Internal validity. Regarding internal validity, we acknowledge that LLMs can be sensitive to format or structure of the prompt [49], or even the order of the few-shot examples [21]. To address this, we manually refine the prompts and ensure that they have the same level of detail, for example, by removing examples that may be described in the original prompt to not impact the few-shot analysis. We also used a fixed order of the prompt techniques that we believe represents a natural sentence flow, and we ensured to use it consistently across all prompts. In addition, the manual creation of few-shot examples for our dataset may have introduced a degree of subjectivity. We therefore involve the first three authors in the process, allowing them to discuss possible examples and select the two most representative ones. Furthermore, to avoid failures due to incorrect function names, we replace the generated signature with the correct one before testing, though failures due to incorrect parameter names may still occur. Finally, since we used recent LLMs, they may have been trained on the same open-source GitHub code we used. To reduce this risk, we avoided code-focused models like Codex and CodeLlama, which are known to be trained specifically on GitHub data. We also checked for memorization by following the method from Schäfer et al. [51] using the *maximum similarity* metric. Similarity was found to produce more meaningful results when identifying memorization compared to other techniques that focus solely on code structure [52]. However, it may still miss other forms of memorization, particularly those involving structural overlap. We found that for all models, 85% of the generated functions had a maximum similarity score below 0.4, and none were higher than 0.7. This suggests that the models produced solutions based on understanding the input, not memorizing training data.

Construct validity. The representativeness of the benchmark (including generation tasks and functions) is an important aspect of construct validity. While current benchmarks often include functions that are not as complex as real-life tasks, we used the CoderEval dataset based on large open-source projects to minimize this threat. However, there remains the question of whether CoderEval fully captures the complexity and diversity of real-world development tasks.

Conclusion validity. For conclusion validity, we focused on three key criteria: similarity, correctness, and quality. While others, like efficiency, could be considered, we argue these suffice for our research questions. Robustness is ensured with multiple metrics for each criterion.

VII. Conclusion

In this study, we have investigated the impact of different prompt techniques on code generation, specifically function synthesis, along three quality dimensions (correctness, similarity to a human-written baseline, and code quality). We studied five prompt techniques, namely few-shot learning, automatic chain-of-thought, providing a persona, providing a signature, and listing packages. We conduct a full factorial analysis of these five factors using CodePromptEval dataset, which we developed based on CoderEval. We studied three current-generation LLMs, namely GPT-4o, Llama3, and Mistral.

Our key lessons learned were that the impact of prompt techniques on correctness, similarity, and quality was not as large as might be expected. Most combinations of prompt techniques do not lead to statistically significant improvements (or regressions) in correctness, quality, or similarity. Providing type information for the function that is to be generated, either explicitly through a signature, or implicitly via few-shot examples, has the most clear positive effect, particularly on correctness. Some prompt techniques have a positive impact on correctness, and others on quality. However, the obvious idea of combining them usually improves neither.

A possible future extension of our research is to evaluate to what extent our findings generalize to other code generation tasks (e.g., line completion, program repair, or the generation of full applications) and other prompt techniques. It is plausible that some of the prompt techniques that did not show a meaningful positive impact on correctness in our experiments (e.g., chain-of-thought) turn out to be more relevant if the generation task is more complex. Additionally, there are other quality metrics, such as performance or energy efficiency, which should be studied in future work — particularly given that recent work indicates that AI-generated code frequently exhibits performance regressions [53].

References

[1] S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, "The programmer's assistant: Conversational interaction with a large language model for software development," in *Proc. 28th Int. Conf. Intell. User Interfaces (IUI)*, New York, NY, USA: ACM, 2023, pp. 491–514.

[2] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2022, pp. 39–51.

[3] R. Khojah, M. Mohamad, P. Leitner, and F. G. de Oliveira Neto, "Beyond code generation: An observational study of ChatGPT usage in software engineering practice," *Proc. ACM Softw. Eng.*, vol. 1, Jul. 2024, pp. 1819–1840.

[4] J. D. Weisz et al., "Better together? An evaluation of AI-supported code translation," in *Proc. 27th Int. Conf. Intell. User Interfaces (IUI)*, New York, NY, USA: ACM, 2022, pp. 369–391.

[5] S. Zheng, J. Huang, and K. C.-C. Chang, "Why does ChatGPT fall short in providing truthful answers?" 2023, *arXiv:2304.10513*.

[6] J. White et al., "A prompt pattern catalog to enhance prompt engineering with ChatGPT," 2023, *arXiv:2302.11382*.

[7] A. J. Fiannaca, C. Kulkarni, C. J. Cai, and M. Terry, "Programming without a programming language: Challenges and opportunities for designing developer tools for prompt programming," in *Proc. Extended Abstracts CHI Conf. Human Factors Comput. Syst. (CHI EA)*, New York, NY, USA: ACM, 2023, pp. 1–7.

[8] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2024, pp. 1–13.

[9] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," in *Proc. Extended Abstracts CHI Conf. Human Factors Comput. Syst. (CHI EA)*, New York, NY, USA: ACM, 2021, pp. 1–7.

[10] H. Yu et al., "CoderEval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2024, pp. 1–12.

[11] S. Barke, M. B. James, and N. Polikarpova, "Grounded Copilot: How programmers interact with code-generating models," *Proc. ACM Program. Lang*, vol. 7, Apr. 2023, pp. 85–111.

[12] K. Ronanki, C. Berger, and J. Horkoff, "Investigating ChatGPT's potential to assist in requirements elicitation processes," in *Proc. 49th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, 2023, pp. 354–361.

[13] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Softw. Eng.*, vol. 50, no. 4, pp. 911–936, Apr. 2024.

[14] H. Li, C.-P. Bezemer, and A. E. Hassan, "Software engineering and foundation models: Insights from industry blogs using a jury of foundation models," 2024, *arXiv:2410.09012*.

[15] R. Tóth, T. Bisztray, and L. Erdődi, "LLMs in web development: Evaluating LLM-generated PHP code unveiling vulnerabilities and limitations," in *Proc. Int. Conf. Comput. Saf., Rel., Secur.*, New York, NY, USA: Springer, 2024, pp. 425–437.

[16] 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," in *Proc. Adv. Neural Inform. Process. Syst.* (A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, eds.), vol. 36, Red Hook, NY, USA: Curran Associates, Inc., 2023, pp. 21558–21572.

[17] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A systematic survey of prompt engineering in large language models: Techniques and applications," 2024, *arXiv:2402.07927*.

[18] G. Wang et al., "Do advanced language models eliminate the need for prompt engineering in software engineering?" 2024, *arXiv:2411.02093*.

[19] T. B. Brown et al., "Language models are few-shot learners," *Adv. Neural Inf. Process. Syst.*, vol. 33, pp. 1877–1901, 2020.

[20] K. Margatina, T. Schick, N. Aletras, and J. Dwivedi-Yu, "Active learning principles for in-context learning with large language models," in *Findings of the Association for Computational Linguistics: EMNLP 2023* (H. Bouamor, J.Pino, K. Bali, eds.), (Singapore), Association for Computational Linguistics, Dec, 2023, pp. 5011–5034.

[21] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, "Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity," *2021*, *arXiv:2104.08786*.

[22] T. Wang, N. Zhou, and Z. Chen, "Enhancing computer programming education with LLMs: A study on effective prompt engineering for Python code generation," *arXiv:2407.05437*.

[23] D. Shrivastava, H. Larochelle, and D. Tarlow, “Repository-level prompt generation for large language models of code,” in *Proc. 40th Int. Conf. Mach. Learn.* (A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, eds.), vol. 202, PMLR, Jul. 2023, pp. 23–29.

[24] Y. Dong, X. Jiang, Z. Jin, and G. Li, “Self-collaboration code generation via ChatGPT,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, pp. 1–38, Sep. 2024.

[25] I. D. Fagadau, L. Mariani, D. Micucci, and O. Riganelli, “Analyzing prompt influence on automated method generation: An empirical study with copilot,” in *Proc. 32nd IEEE/ACM Int. Conf. Program Comprehension (ICPC)*, New York, NY, USA: ACM, 2024, pp. 24–34.

[26] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” in *Proc. Adv. Neural Inform. Process. Syst.*, (S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, eds.), vol. 35, Red Hook, NY, USA: Curran Associates, Inc., 2022, pp. 22199–22213.

[27] Y.-M. Tseng et al., “Two tales of persona in LLMs: A survey of role-playing and personalization,” 2024, pp. 16612–16631.

[28] M. Chen et al., “Evaluating large language models trained on code,” 2021, pp. 1–13.

[29] S. Ren, et al., “CodeBLEU: A method for automatic evaluation of code synthesis,” 2020, *arXiv:2009.10297*.

[30] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2021, pp. 8696–8708.

[31] X. Jiang et al., “Self-planning code generation with large language models,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, Sep. 2024, pp. 1–30.

[32] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin, “AceCoder: An effective prompting technique specialized in code generation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, pp. 1–26, Nov. 2024.

[33] D. Xu et al., “Does few-shot learning help LLM performance in code synthesis?” 2024, *arXiv:2412.02906*.

[34] J. Wei et al., “Chain-of-thought prompting elicits reasoning in large language models,” in *Proc. Adv. Neural Inform. Process. Syst.* (S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, eds.), vol. 35, Red Hook, NY, USA: Curran Associates, Inc., 2022, pp. 24824–24837.

[35] J. Wei, S. Kim, H. Jung, and Y.-H. Kim, “Leveraging large language models to power chatbots for collecting user self-reported data,” in *Proc. ACM Human-Comput. Interact.*, vol. 8, no. CSCW1, pp. 1–35, Apr. 2024.

[36] J. Austin, et al., “Program synthesis with large language models,” 2021, *arXiv:2108.07732*.

[37] B. Athiwaratkun, et al., “Multi-lingual evaluation of code generation models,” 2023, *arXiv:2210.14868*.

[38] R. Khojah, F. G. de Oliveira Neto, M. Mohamad, P. Leitner, “CodePromptEval,” 2024. Accessed: Jul. 1, 2025. [Online]. Available: https://github.com/icetlab/CodePromptEval

[39] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A survey on large language models for code generation,” 2024, *arXiv:2406.00515*.

[40] S. Thakur et al., “VeriGen: A large language model for verilog code generation,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 29, pp. 1–31, Apr. 2024.

[41] V. Corso, L. Mariani, D. Micucci, and O. Riganelli, “Generating Java methods: An empirical assessment of four AI-based code assistants,” in *Proc. 32nd IEEE/ACM Int. Conf. Program Comprehen. (ICPC)*, New York, NY, USA: ACM, 2024, pp. 13–23.

[42] A. Eghbali and M. Pradel, “CrystalBLEU: Precisely and efficiently measuring the similarity of code,” in *Proc. 37th IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE)*, New York, NY, USA: ACM, 2023, pp. 1–12.

[43] I. Heitlager, T. Kuipers, and J. Visser, “A practical model for measuring maintainability,” in *Proc. 6th Int. Conf. Qual. Inf. Commun. Technol. (QUATIC)*, 2007, pp. 30–39.

[44] G. A. Campbell, “Cognitive complexity: An overview and evaluation,” in *Proc. Int. Conf. Techn. Debt.*, New York, NY, USA: ACM, 2018, pp. 57–58.

[45] M. Fowler, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 2018, pp. 95–112.

[46] A. Vargha and H. D. Delaney, “A critique and improvement of the CL common language effect size statistics of McGraw and Wong,” *J. Educ. Behav. Statist.*, vol. 25, no. 2, pp. 101–132, 2000.

[47] T. Ahmed and P. Devanbu, “Multilingual training for software engineering,” in *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2022, pp. 1443–1455.

[48] M. Zheng, J. Pei, L. Logeswaran, M. Lee, and D. Jurgens, “When "A Helpful Assistant” is not really helpful: Personas in system prompts do not improve performances of large language models,” in *Findings of the Association for Computational Linguistics: EMNLP* (Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, eds.), Miami, Florida, USA: Association for Computational Linguistics, Nov., 2024, pp. 15126–15154.

[49] M. Sclar, Y. Choi, Y. Tsvetkov, and A. Suhr, “Quantifying language models’ sensitivity to spurious features in prompt design or: How I learned to start worrying about prompt formatting,” in *Proc. 12th Int. Conf. Learn. Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=RIu5lyNXjT

[50] D. OBrien, S. Biswas, S. M. Imtiaz, R. Abdalkareem, E. Shihab, and H. Rajan, “Are prompt engineering and TODO comments friends or foes? An evaluation on GitHub Copilot,” in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2024, pp. 1–13.

[51] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “An empirical evaluation of using large language models for automated unit test generation,” *IEEE Trans. Softw. Eng.*, vol. 50, no. 1, pp. 85–105, Jan. 2024.

[52] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, “CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models,” in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE),* 2023, pp. 919–931.

[53] S. Li, Y. Cheng, J. Chen, J. Xuan, S. He, and W. Shang, “Assessing the performance of AI-generated code: A case study on GitHub Copilot,” in *Proc. 35th IEEE Int. Symp. Softw. Rel. Eng. (ISSRE)*, Piscataway, NJ, USA: IEEE Press, Jan. 2024, pp. 85–105.

Ranim Khojah is received the Licentiate of Philosophy degree in computer science and engineering from Chalmers University of Technology. She is currently working toward the Ph.D. degree with Chalmers University of Technology and the University of Gothenburg, Sweden. Her research interests include human-chatbot interactions in software engineering.

Francisco Gomes de Oliveira Neto received the Ph.D. degree in computer science from the Universidade Federal de Campina Grande (UFCG), Brazil. He is an Associate Professor in software engineering with the University of Gothenburg and Chalmers University of Technology, Sweden. His main research areas are automated software testing, and (AI) bots to aid software engineers.

Mazen Mohamad (Member, IEEE) received the Ph.D. degree in software engineering from the University of Gothenburg. He is a Researcher with RISE, the Research Institutes of Sweden and a Lecturer with Chalmers University of Technology. His research focuses on security assurance, combined safety and security analysis, AI in software engineering, and AI for cybersecurity.

Philipp Leitner received the Ph.D. degree in business informatics from TU Vienna, Austria. He is an Associate Professor of software engineering with Chalmers University of Technology and the University of Gothenburg, Sweden. His research interests are in empirical software engineering, with a focus on software performance optimization and the development of web- and cloud-based systems. He is a member of the ACM.