# Generative Artificial Intelligence for Industry: Opportunities, Challenges, and Impact

Barun Kumar Saha
Grid Automation R&D
Hitachi Energy, Bangalore 560048, India
barun.kumarsaha@hitachienergy.com

***Abstract*—The recent advances in Generative Artificial Intelligence (GenAI) and Large Language Models (LLMs) have generated significant interest across the world. For a successful adoption of GenAI and LLMs by industry, it is critical to identify their potential benefits, impact, and challenges. Accordingly, in this work, we investigate a few use cases of LLMs, which are relevant across most industry segments. In order to empirically evaluate the impact of GenAI on the code generation use case, we build CodePrompt, a handcrafted dataset of sequential prompts used by a human user to generate code. We approximate efficiency by considering the ratio of the number of tokens of code generated by an LLM to the number of tokens in the user's prompt. Experimental results reveal that a sequential trial of prompts for code generation may lead to an efficiency factor of about 6.33, on average, which means that a user's effort is reduced to about one-sixth.**

***Index Terms*—Artificial Intelligence, Generative AI, Large Language Models, Code Generation, Tokens, Data Pipeline, Bid Engineering**

## I. Introduction

Recently, GenAI and LLMs have received significant attention from researchers, software developers, and users. In general, GenAI generates content—text, image, audio, video, or a combination—based on the instructions provided by the users. In contrast to traditional AI, the scope of GenAI is significantly vast. For example, traditional language models are typically focused on a single task, such as intent and entity identification [1]. Contemporary foundation LLMs, on the other hand, are trained on volumes of heterogeneous data as well as instruction-following datasets [2], which enable them to generate almost any kind of content in response to a user's prompt. Consequently, GenAI and the foundation LLMs, such as Pathways Language Model 2 (PaLM 2) [3], Generative Pre-trained Transformers 3.5 and 4 [4], and Llama 2 [5], are expected to find potential use across different domains.

Although contemporary works [6]–[8] discuss the potential use of GenAI and LLMs in different contexts, there is largely a lack of investigation into the industry-specific use cases. Motivated by this, in this work, we discuss a few industrial use cases where LLMs can play a significant role. The use cases are illustrated with real examples, allowing the stakeholders to identify how they work and what changes need to be made. We also discuss some of the related challenges. The scope of this work is limited to the use cases involving LLMs.

In addition, we empirically evaluate the "impact" of GenAI on a particular use case, code generation, by comparing the average "effort" required by a human user versus GenAI. Such an impact analysis is usually important in the industrial context, which enables a business to invest in the technology. To this end, we build CodePrompt, a handcrafted dataset. CodePrompt is different from some of the contemporary GenAI-generated code evaluation datasets, such as HumanEval [9], LLMSecEval [10], and others [11], in several ways. First, CodePrompt captures the sequential nature of editing and refining the initial prompt to obtain a correct output. Second, CodePrompt contains diverse kinds of coding-related problems, such as test case generation and debugging. Third, the complexity of some of the problems in CodePrompt is fairly high, such as building an application using external API calls. Fourth, not all the problems in CodePrompt have a correct output, which reflects the real-life scenario that solving complex problems may need more effort.

The specific contributions of this work are as follows:

- Identifying the benefits of using LLMs in the industrial context related to bid engineering, data pipeline, and code generation use cases. In addition, reviewing some of the related challenges.
- Building CodePrompt, a handcrafted dataset of prompts—and solutions—with sequential trials used to generate code using PaLM 2.
- Measuring the efficiency of effort required to generate code using GenAI as the ratio of the number of tokens generated to the number of tokens used in the prompt(s). In addition, evaluating the CodePrompt dataset using the aforementioned metric.

The remainder of this work is organized as follows: Sec. II discusses the different use cases and related challenges. Sec. III investigates the efficiency of code generation using LLMs. Finally, Sec. IV concludes this work.

## II. Industrial Use Cases of Generative AI

In this section, we take a detailed look at some of the use cases where GenAI can find potential applications in the industry.

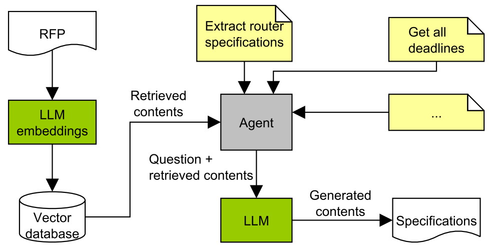

Fig. 1: Illustration of technical and non-technical specifications extraction from an RFP using an LLM. The contents of the document are converted into embeddings (obtained from the penultimate layer of a pre-trained deep neural network) and stored in a vector store. The LLM generates a response based on the original prompts as well as the relevant contents retrieved from the vector store.

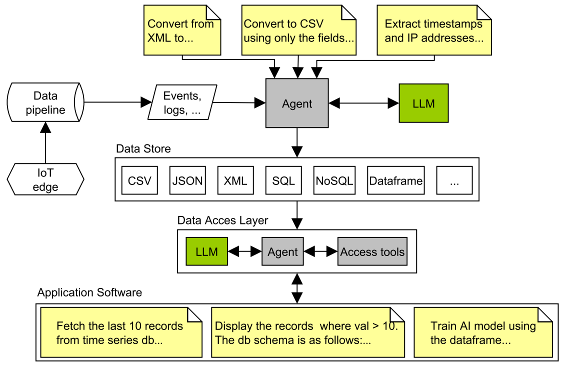
Convert from XML to...
Convert to CSV using only the fields...
Extract timestamps and IP addresses...
Data pipeline
Events, logs, ...
Agent
LLM
IoT edge
Data Store
CSV
JSON
XML
SQL
NoSQL
Dataframe
...
Data Acces Layer
LLM
Agent
Access tools
Application Software
Fetch the last 10 records from time series db...
Display the records where val > 10. The db schema is as follows:...
Train AI model using the dataframe...

Fig. 2: Illustration of an IoT edge-to-cloud data pipeline, where an LLM facilitates data conversion and access.

## *A. Data Extraction and Conversion*

LLMs are trained on large volumes of diverse sets of data. Consequently, LLMs can help largely in extracting information and transforming data into desired formats. Here, we discuss two scenarios where the extractive and transformative features of GenAI and LLMs can be potentially useful.

*1) Bid Engineering:* In Intent-based Engineering [12], an AI-based automated pipeline is envisaged that covers all the engineering aspects of a project, starting from bidding to installation and operations. In the bid engineering phase, vendors prepare bids based on customers' Request for Proposals (RFPs). RFPs are unstructured documents, can contain up to hundreds of pages, and contain both technical and non-technical specifications, among others, for the target projects. Conventional bid engineering involves users manually going through such large documents and copying the requirements in some other place in an organized manner. Co-creation with GenAI can potentially enable the users to simplify this step.

As an example, let us consider a fictional RFP for commissioning a backbone network. Such an RFP may contain technical specifications related to routers, switches, junction boxes, and other components. Here, one can use the contents of the RFP together with an LLM for Retrieval-Augmented Generation (RAG) [13] of contents. A series of questions (for example, "What are the different networking-related components required for this project?") may be asked, whose answers would be retrieved by an LLM based on the provided document. Subsequent queries may be aimed at retrieving the other relevant information, such as documents to be submitted. Figure 1 illustrates such a workflow using LLMs.

In contrast to training an AI model from scratch, a GenAI-based approach toward bid engineering can save significant effort and time. However, it may be noted that the contents of an RFP may largely vary from another. Therefore, a single sequence of LLM prompts may not work for every RFP. In this regard, a manual pre-screening of an RFP may provide an idea about what questions to ask the LMM.

*2) Data Pipeline:* The Industrial Internet of Things (IIoT) typically uses a data pipeline to collect data from field devices, move them to the cloud, store them in a database, and perform analysis on such stored data. In practice, a real-life data pipeline contains, apart from the basic data transportation functionality, several other software modules, for example, information schema, pipeline health monitoring, and APIs. Finally, several downstream applications feed on the collected data, for example, to train an AI model and display statistics in a dashboard. A key point to observe here is that, across the data pipeline, the same data elements are often moved to different places and stored or accessed in different formats. LLMs can potentially play a key role here in simplifying the data pipeline architecture.

Figure 2 illustrates a high-level architecture of an IoT data pipeline, where data from the edge, for example, a substation, is transported to the cloud. Here, an LLM plays a key role, providing the universal data conversion and access functionality. The figure also shows an AI agent, which can plan for the appropriate tasks. In addition, such an agent potentially has access to different external tools, such as database engines, which allow it to achieve the tasks.

When new data becomes available via the pipeline, appropriate prompts can be set to convert and store data in one or more desired formats. For example, a downstream data pipeline health analytics application may need to store the timestamps and the number of bytes received in a relational database, which can be queried using the Structured Query Language (SQL). Recent works indicate that SQL queries can be generated from text with high accuracy [14], [15]. On the other hand, a different module to train an AI model may require that new data points be appended in a flat file.

To illustrate, let us consider Listing 1, where an LLM is asked to convert JSON data into CSV by retaining only a few fields. Accordingly, the output data can be generated, as shown in Listing 2. An agent can save these data in an external file.

Listing 1: LLM prompt and the original JSON data

```
Given some input JSON data, convert them to CSV format. The
    CSV columns should include timestamp, objectId,
    variable, and value.

Data:
{
        "data": [
        {
                "timestamp": "2022-06-22T10:17:02.457",
                "model": "device",
                "objectId": "af9a8f00-55e9-40aa",
                "processed": "2022-09-13T13:03:01.806",
                "tenantId": "52c21c00-bbb4-4234",
                "value": "0",
                "variable": "altitude"
        },
        {
                "timestamp": "2022-06-22T10:17:02.467",
                "model": "device",
                "objectId": "af9a8f00-55e9-40aa",
                "processed": "2022-09-13T13:03:02.816",
                "tenantId": "52c21c00-bbb4-4234",
                "value": "2.2.5",
                "variable": "hardwareRevision"
        }
}
```

Listing 2: Transformed data in CSV format

```
timestamp,objectId,variable,value
2022-06-22T10:17:02.457,af9a8f00-55e9-40aa,altitude,0
2022-06-22T10:17:02.467,af9a8f00-55e9-40aa,hardwareRevision
    ,2.2.5
```

A complementary aspect of the above-discussed scenario, as shown in Figure 2, is the use of LLMs as a data access agent. For example, if one wants to find all the events reported in a given time interval from relational and non-relational data stores, one would need to write two queries with different syntax, although the semantic purpose of both queries is equivalent. In contrast, with GenAI, one can prompt an LLM with the data store name and the query written in natural language. In response, a suitable query specific to the data store can be generated and executed to fetch the results.

To summarize, LLMs can potentially play the role of a data abstraction layer, simplifying how data are transformed, stored, and accessed. Moreover, the scope of such a layer need not remain limited to a single project but may be used as a service across multiple projects within an organization.

## B. Coding Companion

Given a problem description, LLMs can generate the relevant code, documentation, and test cases. In addition, LLMs can also help troubleshoot problems, such as identifying errors in the code and suggesting appropriate code fixes.

Figure 3 illustrates this idea at a high level, where a user uses an Integrated Development Environment (IDE) to develop software. A user can type in instructions (for example, “Write Python code to. . . ”), which are sent to an LLM via a plugin installed with the IDE. When the LLM returns the code response, the plugin, in turn, can insert the code snippet at an appropriate location in the code editor. An advanced plugin may also allow users to select a block of existing code from the editor and send that as a context to the LLM so that the new code is generated based on the existing code. The IDE-centric workflow illustrated in the figure may help users to easily work with diverse and complex scenarios. For example, in a scenario where the execution of code in the IDE results in an error, a user can select the concerned error message and ask the LLM for a solution.

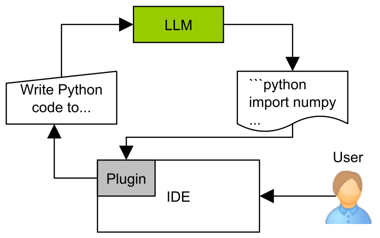

Fig. 3: Code generation in an IDE (or code editor) using an LLM, potentially facilitated by a plugin.

## C. Other Use Cases

GenAI has potential volumes of other use cases suitable for different industries. GenAI can find significant use in the sales, marketing, and customer support departments, among others. For example, sales pitch emails can be customized to match different personas. On the other hand, a RAG-based approach combined with an internal knowledge base may help to respond to common or typical queries raised by customers, for example, how to fix a certain problem.

In general, industrial adoption of and co-creation with GenAI offers two significant benefits. On the one hand, customers can obtain relatively quick and appropriate resolutions to their problems, improving customer satisfaction. On the other hand, with common and repetitive tasks handled by GenAI, employees can focus on more important activities, potentially improving the services offered by an organization.

## D. Challenges of Using Generative AI

Before concluding this section, we take a look at some of the challenges related to LLMs and GenAI.

*1) Output Verification:* LLMs can sometimes suffer from the hallucination effect, where the seemingly good output may be incorrect. Therefore, content generated by AI should be properly checked for correctness and other related aspects. For example, source code generated by AI, similar to user-written code, should be subjected to static and runtime verification. Moreover, when LLMs are used to generate database queries, it should be verified that the generated queries, especially if they are complex, perform the desired task. In addition, in the case of content retrieval using LLMs, the precision and recall may also need to be assessed. For example, when an LLM is used in bid engineering, it is expected to retrieve all the specifications relevant to the target category. Failing to retrieve any requirement in such a scenario may have a negative effect on the bid prepared based on LLM-extracted information.

*2) Prompt and Response Testing:* A prompt submitted to an LLM usually contains the problem description, a context, and the output from the previous interaction. In general, different LLMs perform differently in terms of content generation. Moreover, given a specific LLM, the performance may also vary among its different versions or sizes of the models. In essence, one might think of an LLM as a stochastic algorithm. Therefore, the output that an LLM produces may vary based on the input (i.e., prompt) as well as other hyperparameters, such as the temperature of the model. In other words, as illustrated in the previous section, prompt engineering—crafting a suitable set of instructions—is an important aspect, often requiring some experimentation. Therefore, as LLM-based applications continue to grow, in the future, a significant effort may be directed toward prompt engineering and writing test cases to verify the behavior of such applications.

*3) Operational Cost and Limits:* Open Source LLMs can be potentially run on the cloud or on-premises. Proprietary LLMs, on the other hand, are only available via the respective service providers. However, in both scenarios, a significant computing infrastructure, and therefore, a huge investment, is required. In addition to the cost, LLMs often have upper limits on the number of tokens that can be used in input and output. Consequently, text from large documents, such as RFPs, may not be provided to an LLM in a single attempt but may need some buffering. As noted earlier, missing any information from any RFP is unacceptable in bid engineering. In addition, the APIs have certain usage limits, for example, 100 calls may be made per minute at most. Therefore, LLMs-based software also needs to take such constraints into account.

## III. Impact of Generative AI

In this section, we consider a specific use case of GenAI—code generation—and empirically evaluate its impact. Our objective is to roughly estimate how much effort can be saved when developers use GenAI to develop software.

### *A. Efficiency Measurement Based on Tokens Count*

Let $P$ be a set of programming-related problems (equivalently, questions, instructions, or prompts). When a prompt $P_i \in P$ is submitted to an LLM, a response text $C_i$ is generated.

In an ideal scenario, $C_i$ represents an accurate and complete solution that a user "intends" to have in response to a prompt $P_i$. However, in practice, crafting a prompt for LLMs may require some experimentation. In other words, users may need to make minor or major changes in the prompts to obtain the desired output. Accordingly, for any prompt $P_i \in P$, let $P_i^j$ be the $j$th trial (equivalently, attempt or refinement) for the prompt $i$, where $1 \leq j \leq \tau_{max}$ is an integer and $\tau_{max}$ denotes the maximum number of prompt alterations that the user can attempt. Moreover, let $\tau_i$ be the number of trials made for any prompt $P_i$.

We are interested in empirically estimating how much impact can generative AI have in the context of code generation. Here, an empirical measure of the "impact" can be obtained by considering the average number of tokens a user is saved from typing. We consider tokens as a unit rather than the source lines of code or other alternatives primarily due to two reasons. First, since different programming languages have different syntax, the average length of a line of code, and therefore, the effort required to type that in, would vary. Second, since the input prompts consist of a mix of both code and natural language text, tokens offer a uniform approach for normalizing the size of input and output.

Let $T$ be a tokenizer so that, for any input text $x$, $T(x)$ represents a list of tokens obtained from $x$ and $|T(x)|$ denotes the number of tokens. Then, the effort gained (or saved) in terms of the number of tokens for any given prompt $P_i^j$ and its code response $C_i^j$ is measured as:

$$\eta_i^j = \frac{|T(C_i^j)|}{|T(P_i^j)|}. \tag{1}$$

Accordingly, the average token efficiency of a code prompts dataset (including the original prompts and their $j$ trials) becomes:

$$\eta = \frac{1}{n} \sum_i \sum_j \eta_i^j, \tag{2}$$

where $n = \sum_{k=1}^{|P|} \tau_k$.

The baseline efficiency measures in (1) and (2) can be improved by considering the "effective" prompt inputs and their sequential statefulness. To understand this, let us recall that any prompt may contain both natural language text and code. It is expected that the text in the prompts is written by the users themselves. However, the code snippet may already exist so that a user merely *copies* that code in the prompt rather than manually writing it in the prompt. Such code may be previously generated using AI. Alternatively, such code may have been previously written by a user, but not for the purpose of using it with a code prompt. In addition, error messages—from run-time or compile-time errors—may also be copied in a prompt. In other words, the "effective" volume of text (or tokens) written by a user in a prompt may be less than the length of the prompt's text. Accordingly, let $\tilde{P}_i^j$ be the effective prompt so that $|\tilde{P}_i^j| \leq |P_i^j|$. Then, (2) can be refined as:

$$\tilde{\eta} = \frac{1}{n} \sum_{i,j} \tilde{\eta}_i^j = \frac{1}{n} \sum_{i,j} \frac{|T(C_i^j)|}{|T(\tilde{P}_i^j)|}. \tag{3}$$

It may be noted that, given a prompt and its $j$ trials (or sequential refinements), $P_i^1, P_i^2, \cdots, P_i^j$, any prompt in the sequence is typically obtained by adding, removing, or substituting texts (tokens) in the previous prompt. For any two sequences of tokens $x$ and $y$, let $\delta(x, y)$ be the edit distance between $x$ and $y$. In other words, one requires $\delta(x, y)$ editing operations to produce $y$ from $x$. Accordingly, the tokens count of an effective prompt $\tilde{P}_i$ with $\tau_i$ trials, $\forall i$, considering its possible sequential attempts, is measured as:

$$\kappa(\tilde{P}_i) = \begin{cases} |\tilde{P}_i|, & \text{if } \tau_i = 1 \\ |\tilde{P}_i^1| + \sum_{k=2}^{\tau_{max}} \delta(\tilde{P}_i^{k-1}, \tilde{P}_i^k), & 2 \leq \tau_i \leq \tau_{max} \end{cases} \tag{4}$$

The average efficiency, considering the sequential dependence of the effective prompts and the final version of the code produced in the sequence of trials, is expressed as:

$$\eta^* = \frac{1}{|P|} \sum_i \frac{|T(C_i^{\tau_i})|}{\kappa(\tilde{P}_i)}. \tag{5}$$

In other words, $\eta^*$ in (5) indicates the multiplicative scale with which generative AI generates code-related tokens, on average, for every single token written by a user. Therefore, code generation using GenAI is useful only when $\eta^* > 1$.

TABLE I: Summary of the CodePrompt dataset. The “Correct” field indicates whether or not the correct output was generated.

| ID | Type | Max. Trials | Correct | ID | Type | Max. Trials | Correct | ID | Type | Max. Trials | Correct |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | write | 3 | no | 11 | complete | 3 | no | 21 | write | 1 | |
| 2 | write | 1 | | 12 | complete | 1 | | 22 | write | 2 | |
| 3 | write | 3 | no | 13 | complete | 1 | | 23 | write | 1 | |
| 4 | complete | 1 | | 14 | fix | 3 | | 24 | write | 1 | |
| 5 | complete | 1 | | 15 | question | 1 | | 25 | complete | 3 | no |
| 6 | complete | 1 | | 16 | question | 1 | | 26 | write | 1 | |
| 7 | write | 3 | no | 17 | fix | 3 | | 27 | write | 3 | |
| 8 | complete | 3 | | 18 | question | 1 | | 28 | write | 1 | |
| 9 | fix | 2 | | 19 | fix | 1 | | 29 | write | 1 | |
| 10 | write | 1 | | 20 | write | 3 | no | 30 | write | 3 | no |

### B. *Dataset and Performance Evaluation*

We built CodePrompt[1], a handcrafted dataset, in order to empirically evaluate the impact or usefulness of GenAI for code generation. CodePrompt consists of 30 (i.e., $|P| = 30$) programming, AI, and software development-related problems as well as their solutions generated using PaLM 2. Each problem consists of three prompts at most (i.e., $\tau_{max} = 3$), representing the scenario where a user makes a maximum of three attempts to have a solution generated by AI. Collectively, there are 54 prompts, and therefore, 54 code responses. The problems or prompts in the dataset primarily relate to three different categories—write code from scratch based on natural language instructions, complete a partially provided code snippet, and fix problems in the code. As shown in Table I, about half of the problems from the CodePrompt dataset required more than one attempt to solve.

CodePrompt used the code-bison-32k model of PaLM 2 to generate solutions. The temperature was set to 0.01. A maximum of 4096 tokens were allowed in the output. PaLM 2 was used in a stateless (i.e., non-chat) mode, so the output of the prior interactions did not affect the current output.

We manually executed the code response for each prompt to verify its correctness. Since our objective was to generate a fragment of code, and not an entire software project, we assumed that all necessary modules, files, and libraries required to run such code were already available. Accordingly, we also ignored trivial omissions and variations in the code response, such as not importing a library. The prompts failed to achieve correct solutions for 7 out of the 30 problems. In other words, the CodePrompt dataset achieved about 76.66% correct results using PaLM 2. On the other hand, each problem, on average, used 1.80 prompts.

All prompts and code responses were saved to different files. Subsequently, we used Tiktoken[2], an Open Source tokenizer based on Byte Pair Encoding, to tokenize texts from the prompts and code, based on which we computed the token counts and the efficiency measures.

### C. *Results*

Figure 4 shows the distribution of token count for all the prompts, the corresponding effective prompts, and the resulting code responses generated by PaLM 2. In general,

[1]https://github.com/barun-saha/CodePrompt

[2]https://github.com/openai/tiktoken

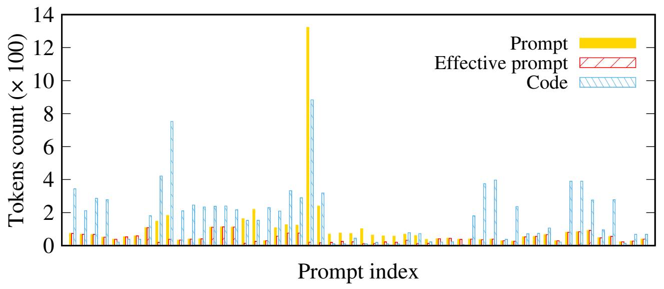

Fig. 4: Tokens count distribution for the prompts, effective prompts, and the resulting code.

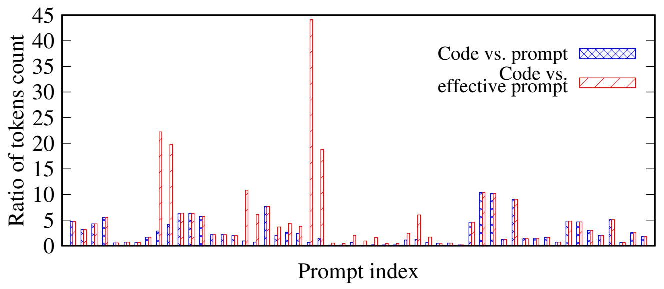

Fig. 5: Code generation efficiency for the prompts and the corresponding effective prompts.

the code generated in response to the prompts contains more tokens. This is evident from the efficiency graphs in Figure 5, where the ratio of tokens count in the code versus the prompts and the effective prompts are shown. Moreover, since the effective prompts usually contain less number of tokens than the original prompts, the latter ratio is higher. In particular, we found the average efficiency, as measured by (1) and (3), to be 2.61 and 4.99, respectively. In other words, for every token in the input prompt, 2.61 times more tokens, on average, are produced in the code. On the other hand, for every token in the effective prompt, 4.99 times more tokens are generated in the code, on average.

Figure 6 shows the distribution of token counts by considering the sequential prompts for the coding problems, as discussed in (4). The corresponding efficiency measure is shown in Figure 7. The boxes with relatively darker shades indicate the problems for which correct code output was obtained. The boxes with relatively lighter shade indicate the problems for which no correct output code was obtained in

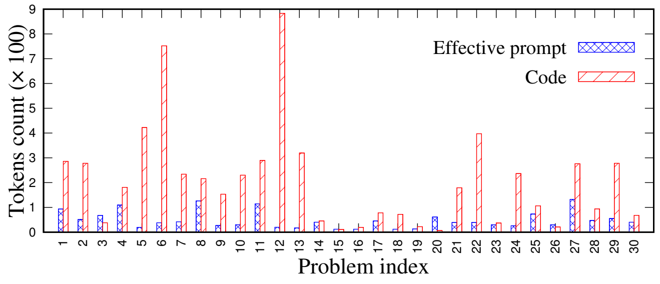

Fig. 6: Tokens count distribution for the problems considering the sequential prompts, effective prompts, and the final version of the code generated.

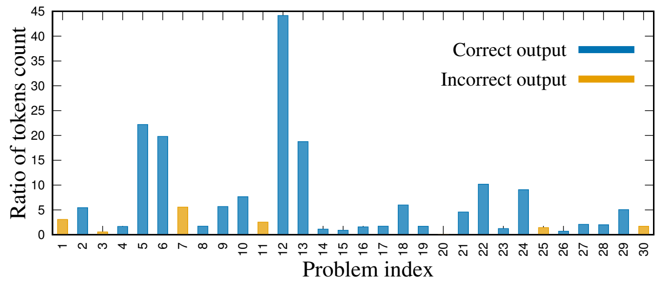

Fig. 7: The efficiency measure, by considering the sequential effective prompts for all the problems from the dataset.

$\tau_{max} = 3$ trials.

In particular, the average efficiency in this case, as measured by (5) was found to be 6.33. In other words, when a user attempts to generate code by providing some instructions and making at most two revisions of the original instructions, the code generated contains 6.33 times more tokens than the input. This indicates that a user, on average, has to spend less effort (and time) in generating code using AI. This, in turn, largely points toward increased efficiency and productivity.

Figure 7 also shows that the efficiency for certain types of code generation problems is higher than the others. For example, Problem 12 asked to generate test cases for a given module of code. Consequently, the effective prompt had a much lower size—there was only one line of text instruction, whereas there were more than 120 lines of code based on which the test cases were to be generated. In contrast, the generated code response had about 100 lines of code. Therefore, the ratio of the tokens count of the response versus the effective prompt was very high. On the other hand, Problem 20 failed to generate code. Consequently, the efficiency was close to zero.

Before closing this section, it may be noted that (1) through (5) do not account for several other factors, such as a user's lack of familiarity with a given technology and the complexity of the code to be written. In general, when users write code, they often need to go through documentation to find the appropriate libraries or API calls, which require additional effort and time. With generative AI, this effort can be reduced, too. Therefore, in practice, the efficiency gained—or the impact that generative AI has on code generation—is expected to be relatively greater than what is reported by the preceding performance evaluation results.

## IV. Conclusion

In this work, we investigated how LLMs can be useful in the industrial context by considering different use cases. We also empirically evaluated the impact of GenAI on code generation by developing the CodePrompt dataset and comparing the sizes of user inputs and AI-generated output. Experimental results revealed that code generation with LLMs can reduce human effort to one-sixth, on average, which indicates that co-creation with GenAI is useful. Finally, we also discussed some of the challenges that GenAI offers.

In the future, this work can be extended in different ways. The CodePrompt dataset can be expanded to include further diversity. Moreover, other industry-relevant use cases can also be investigated.

## References

[1] B. K. Saha, L. Haab, and L. Podleski, "Intent-based Industrial Network Management Using Natural Language Instructions," in *IEEE CONECCT 2022*, Jul. 2022, pp. 1–6.

[2] L. Ouyang *et al.*, "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 27 730–27 744.

[3] R. Anil *et al.*, "PaLM 2 Technical Report," *ArXiv*, vol. abs/2305.10403, 2023. [Online]. Available: https://arxiv.org/abs/2305.10403

[4] OpenAI, "GPT-4 Technical Report," 2023, [Online] Last accessed: Sep. 14, 2023.

[5] H. Touvron *et al.*, "Llama 2: Open Foundation and Fine-Tuned Chat Models," 2023, [Online] Last accessed: Sep. 14, 2023.

[6] V. Bilgram and F. Laarmann, "Accelerating Innovation With Generative AI: AI-Augmented Digital Prototyping and Innovation Methods," *IEEE Engineering Management Review*, vol. 51, no. 2, pp. 18–25, 2023.

[7] I. L. Alberts *et al.*, "Large language models (LLM) and ChatGPT: what will the impact on nuclear medicine be?" *European Journal of Nuclear Medicine and Molecular Imaging*, no. 50, pp. 1549–1552, 2023.

[8] D. Vaz, D. R. Matos, M. L. Pardal, and M. Correia, "Automatic Generation of Distributed Algorithms with Generative AI," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, 2023, pp. 127–131.

[9] M. Chen *et al.*, "Evaluating Large Language Models Trained on Code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[10] C. Tony, M. Mutas, N. Ferreyra, and R. Scandariato, "LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, May 2023, pp. 588–592.

[11] H. Koziolek, S. Gruener, and V. Ashiwal, "ChatGPT for PLC/DCS Control Logic Generation," in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2023.

[12] B. K. Saha, L. Haab, and D. Tandur, "A Natural Language Understanding Approach Toward Extraction of Specifications from Request for Proposals," in *Proceedings of 2023 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, 2023, pp. 205–210.

[13] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474.

[14] B. K. Saha, P. Gordon, and T. Gillbrand, "NLINQ: A natural language interface for querying networkperformance," *Applied Intelligence*, vol. 53, pp. 28 848–28 864, Dec. 2023, DOI: https://doi.org/10.1007/s10489-023-05043-z.

[15] C. J. Borjal *et al.*, "Parallel Corpus Curation for Filipino Text-to-SQL Semantic Parsing," in *2023 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, 2023, pp. 163–169.