# APERFCODE: AUTO CONVERSION TO PERFORMANT CODE

**Sharod Roy Choudhury, Mayank Mishra& Rekha Singhal& Sirish Karande**
TCS Research
Mumbai
{sharod.rchoudhury,mishra.m,rekha.singhal,sirish.karande}@tcs.com

## ABSTRACT

Advanced developers often review code to improve the quality of readability, security, stability, cost, memory and compute performance, etc. The problem becomes harder when the performance of the code depends on the input data. Further, manual review of all the code does not scale well; hence, automation is desired in such a process. Most often, the data-dependent changes required in the code need refactoring, such as replacing a data structure to one amenable to the input data pattern (e.g. 'data frame' with a 'list' if the input data has a large number of repetitive values). This makes the learning task complex and the training data sparse for building a model to learn the conversion of a code to its performant version. We have considered a propriety dataset (data-frame-based DL pipeline pre-processing codebase) from a large MNC. We seek to leverage the general-purpose knowledge and understanding being demonstrated by LLMs like ChatGPT to help automate the process of improving the performant quality of code. We have evaluated our approach against hand accelerated codebase of a proprietary dataset of a large MNC using two methods: 1) where the developer's intent for transformation is known as one exemplar (referred to as 'Domain-Driven') and 2) where examples are not available and the intent has to be specified with a generic natural language instruction (referred to as 'Generic-NL'). We show in our empirical analysis that both our approaches achieve $100\%$ success in accelerating a given slow code snippet. For $95.65\%$ cases, code converted through the 'Domain-Driven' approach is faster than even the hand-accelerated ones. Further, for $56.52\%$ cases, the speedup is $2\times$ more than with hand-accelerated ones.

## 1 MOTIVATION

AI democratization tools enable the automatic building of DL pipelines, where most of the data-preprocessing code is based on Python data frames. The Python data frames perform poorly on data sets with many repetitive values. Hence, such pipelines may incur large pre-processing time, which contributes to $80\%$ of the overall AI/ML model preparation timeHellerstein et al. (2018), Press, Terrizzano et al. (2015). FASCA[1] Mishra et al. (2021) does both static and dynamic analysis of such codes and identifies the various anti-patterns of performance bottlenecks. It further suggests templates for the performant version of the identified anti-patterns. These templates may need a change in the data structure (e.g. from data frame to list) to accelerate, unlike system-level acceleration, which could be automated (e.g. using a large number of cores, faster hardware, and pipelining, etc.). These changes need human intervention to replace the bad code snippet with its performant version.

To democratize performance in AI pipelines, we need to automate the conversion of such pre-processing code to its performant version. There has been significant interest in using neural approaches for program synthesis and code generation tasks. In particular, recent works have looked at the use of Large Language Models (LLMs) through various generations: training a task/dataset-specific custom model Robustfill Devlin et al. (2017), KarelChen et al. (2019)Bunel et al. (2018), models which can be fine-tuned PLBART Ahmad et al. (2021), CodeBERT Feng et al. (2020), CodeGPT Meyrer et al. (2021), CodeT5 Wang et al. (2021), models that are too large to be finetuned

[1]Tool to discover performance anti-patterns with their performant version

and have to be mostly used as black box De la Rosa & Fernández (2022), Pasquini et al. (2010) and jig and models that are supervised to be aligned with human intent as done in ChatGPT. Nevertheless, neural approaches for improving code performance have remained largely unexplored.

The effectiveness of a program or code synthesis system depends on the synthesizer's quality, the quality with which the specifications capture the developer's intent, and finally, the effectiveness of a verification or post-processing system that can be used to guide or filter the output of the synthesizer. In contrast to our experience working with LLMs, we have recently discovered great value in utilizing ChatGPT as the synthesizer. Natural language and Input/Output examples-based approaches are common in specifying the developer's intent. We consider both these approaches.

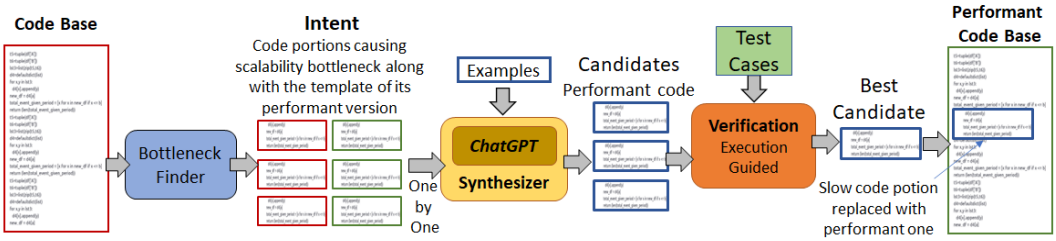


Figure 1: Methodology to generate a performant version of the code using the domain knowledge powered Input-Output examples.

In the general natural language-based approach of specifying the developer's intent (referred to as Generic-NL method), we express the intent as generic natural language instruction, whereas, in the domain knowledge-based approach (referred to as Domain-Driven method), the intent is described by domain knowledge powered input-output[2] examples by a performance expert (refer Figure1). Finally, towards post-processing, we consider two approaches, one where we call the synthesizer to iterate on its suggestion to improve the code quality or choose among multiple candidates by utilizing the execution of some held-out test cases. We observe that both approaches are effectively improving the performant quality of code. However, the Domain-Driven approach can conduct larger modifications in code, leading to significantly greater gains.

We have evaluated our approach against the hand-accelerated codebase of a proprietary dataset of a large MNC. Our empirical analysis shows that both approaches achieve $100\%$ success in accelerating a given slow code snippet. For $95.65\%$ cases, code converted through the 'Domain-Driven' approach is faster than even the hand-accelerated ones. Further, for $56.52\%$ cases, the speedup is $2\times$ more than with hand-accelerated ones.

The contributions of the paper are as follows

- We have proposed APerfCode, an architecture for auto conversion of code to its performant version using ChatGPT
- We have studied two approaches where domain experts can provide intent in the form of Input-Ouput pairs of slow and fast code, respectively, and where the intent is given in generic natural language.
- We have evaluated the approaches on a proprietary dataset of AI pipeline in MNC's recommender system. We have observed $100\%$ success in accelerating the code with better speed up by Domain-driven approach.

The paper is organized as follows. Section 2 discusses the related work. Section 3 describes the problem statement. Section 4 describes the dataset used. The ChatGPT has been described in section 5, and the methodology is described in section 6. Section 7 describes the empirical observation and discussions, and finally, section 8 discusses future work.

[2]The input is the snippet of slow code, and output is the snippet of fast code.

## 2 APERFCODE

We are proposing APerfCode, by which we can automatically convert a slow code snippet to a fast code. We are using ChatGPT to generate accelerated code snippets. We have described ChatGPT in the next section followed by the dataset used. Then we provide a formal description of our problem statement. We further state some LLM-based approaches that we tried before.

#### 2.0.1 CHATGPT

ChatGPT is a conversational deep learning model, or chatbot, developed by OpenAI. It was trained using the Reinforced Learning with Human Feedback (RLHF) method and is based on the GPT3.5 architecture. While OpenAI has noted that the responses generated by ChatGPT may not always be accurate, our exploration has revealed that it is quite effective in the domains of code generation, code summarization, and code transformations.

### 2.1 DATASET USED

The dataset utilized in our study consists of pairs of slow-code and fast-code snippets derived from a real-world machine learning (ML) pipeline deployed for a recommender system in the retail industry. All codes can be divided into 4 types. There were 20 samples of type 1 and then 1 each of the last 3. The code snippets, which included data processing operations such as filtering and grouping using the pandas[3] library, were sourced from the ML pipeline. The slow-code snippets, which were identified as performance bottlenecks (high latency, low throughput) through profiling using tools such as Scalene, were the focus of our study. The profiling process is thoroughly documented in our prior work, FASCA Mishra et al. (2021). The accompanying fast-code snippets in each pair were the hand-optimized versions created by expert practitioners.

We obtained the dataset from internal sources of TCS, so we are unable to share more details. Due to this, the amount of data that we were able to use to test was also limited. Because there is no public dataset for this problem statement.

While the ML pipeline contained numerous performance bottlenecks of varying types, we have chosen to focus on the most pressing ones for this paper.

### 2.2 PROBLEM STATEMENT

Given a piece of code that humans or LLM may write, accelerate it. So we aim to output a version of the slow code that is the fastest and takes the least latency. There is debate about whether an LLM can generate the correct code. But it may not be the fully accelerated version, even though it can. So our work will also be useful in a future where code written by LLMs is commonplace if such a future comes to pass. However, the output code we generate should pass all the input-output test cases and pass validation tests so that the original intention of the slow code is not broken.

### 2.3 LLM BASED APPROACHES

We have experienced the following constraints while applying various LLM-based methods to improve code performance automatically.

(1) Methods based on training task-specific sequence generation models: Such an approach requires a large amount of parallel data or, in the absence of parallel data, a mechanism to invoke methods such as back-translation Bjork & Christopoulos (1998) We are not aware of any suitable large public repositories which provide a parallel corpus of slow and accelerated code.

(2) Methods that use grammar to generate synthetic data: We faced two challenges in adopting such an approach. Grammars that model simple edits to code cannot capture the abstractions that represent large changes in the lexical form of the code. Unlike other examples in literature, Karel Chen et al. (2019)Bunel et al. (2018), Robustfill Devlin et al. (2017) and GPT3 Ahmad et al. (2021) paper, the input and output are code. Therefore one needs to define grammar to generate the input

[3]Pandas is a Python library for data analysis and manipulation. https://pandas.pydata.org/

examples as well. It is hard to handcraft such a grammar to represent the real-world examples we have been working with suitably. Furthermore, it is also hard to induce grammar from a few examples.

(3) Methods that use the knowledge captured by LLMs: Recent approaches Jigsaw jig have looked to leverage the ability of LLMs like GPT-3 and Codex for code generation tasks. In such approaches, there is complete dependence on the LLM doing a good enough job with synthesis so that even when mistakes are made, they can be corrected with some local search and repair. We could not find suitable prompts for GPT-3, GPT-J, or Codex. (4) ChatGPT has been used in Sobania et al. (2023) to evaluate error correction. In Castelvecchi (2022), code generation has been evaluated.

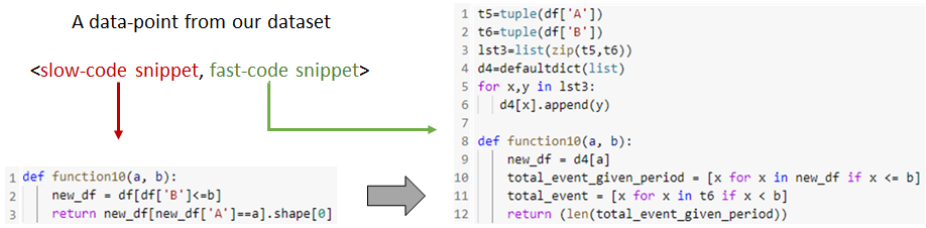


Figure 2: An example of data point from our dataset. On the left is the slow code. On the right is the fast code after the data structure change transformation.

## 3 Our Methodology

To validate the correctness and validity of the code produced by ChatGPT, we conducted verification by manually creating test cases for our dataset. However, for any other dataset that may be used, it is important to ensure the availability of sufficient test cases for verification and validation.

We have experimented on pandas data frame-based codes with filter operations using conditional statements. We have provided an example in Figure 2. But the method discussed does not assume the nature of the input code in any form. So our method is generic and can be applied to any piece of data.

So we restrict our work to Python language only as different languages may have different nuances. Also, we do not inherently use the full code to accelerate using ChatGPT. Rather we use FASCAMishra et al. (2021) first to identify the slow lines of code or slow blocks of code. Then we provide the slow code snippets to the ChatGPT and obtain accelerated versions.

In our dataset[4], all code acceleration was achieved by modifying the data structure. The process involved reading the data into a Pandas data frame, extracting the required columns into tuples, zipping the tuples to form a list, and using the list to create a defaultdict. This defaultdict was then utilized to filter the necessary data. This approach was applied consistently for all data points in our dataset.

To accelerate the code using ChatGPT, we evaluated two methods. The first involved directly asking ChatGPT to perform code acceleration, while the second involved generating the method description outlined above using ChatGPT, which was then used to perform code acceleration.

### 3.1 Generic-NL method

In this method, shown in Figure 3, we used ChatGPT directly to accelerate the code. We provided the slow code and iteratively asked ChatGPT to generate accelerated versions of the same Code. We tried giving the full slow code to ChatGPT, but by doing so, ChatGPT is getting waylaid, and the code thus generated is slower than the input. ChatGPT missed the lines of code that the dataset had targeted to accelerate and was trying to accelerate other lines of code. This happened because ChatGPT has general world knowledge. But we are not providing it with test cases, and hence ChatGPT can not do the profiling of the code by itself. We used the FASCA tool to first profile and

[4]Our dataset consists of tuples in the form of <slow-code, fast-code>.

identify the slow lines of code and then asked ChatGPT to accelerate only the identified slow lines of code. Then we replaced the slow lines of code with the fast lines generated by ChatGPT. This gave us a fast output code. The prompt we used is shown in red boxes of Figure3.

The above prompt would give us one accelerated version of the slow code. But we know that a code can be written in many different ways, each of which will have different latencies. So it is possible to accelerate a slow piece of code using many methods with different latencies. So to emulate this, after getting the output of the 1st prompt, we kept providing the prompt in 3 again and again.

This gave us multiple accelerated codes for the slow input code. We stopped when the last five codes outputted by ChatGPT were a repetition of one of the codes obtained before. This was our stopping criteria. The flowchart for this method is provided in Figure 3. An example of a slow sample snippet with the five fast code snippets obtained in this method is presented in Figure 4

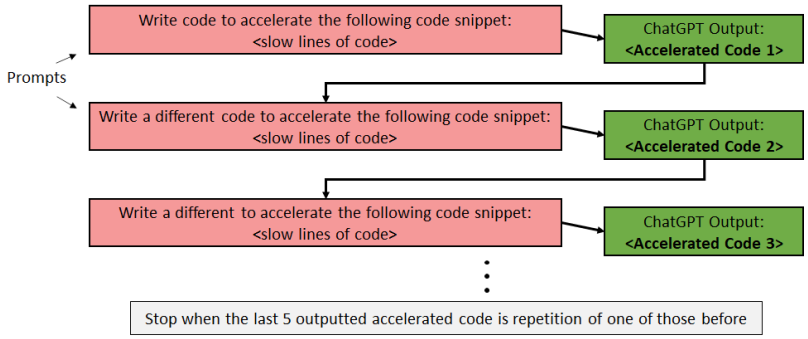


Figure 3: Schematic representation of Generic-NL method

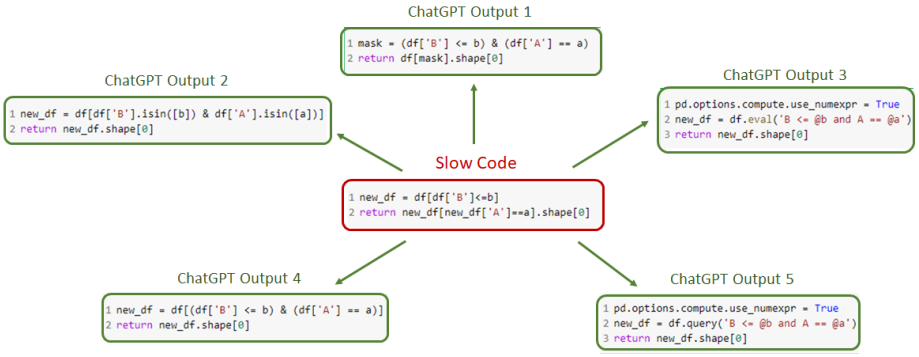


Figure 4: ChatGPT Accelerated snippets using Generic-NL method

### 3.2 Domain-driven method

In this method, we tried to emulate the changes done in the data. So here, data-structure change will happen. We need to describe the transformation and provide the slow code. So ChatGPT would follow the transformation process and produce code at least as fast as the gold dataset. The textual description of the transformation can be mentioned once and then referred to for further transformations and need not be copied in every successive prompt. The prompts can be as in 5.

But then, this transformation text needs to be written by an expert. If a dataset contains many codes with multiple transformation methods, it may take a lot of human effort to create a textual description of each transformation. So we thought we could use ChatGPT itself to generate the transformation text. This we did by providing a single pair of slow code snippets and fast code snippets to ChatGPT. As we see Figure 6, ChatGPT can generate text that accurately describes the

transformation process. The first paragraph creates a description of the slow code. The second paragraph creates a description of the fast code. In the last paragraph, it creates a textual description of the transformation. Now with some small grammatical changes, we can transform it into a prompt that is very similar to the prompt written by a human. An example of the Domain-Driven method is presented in Figure 6. In this method, we found that providing only the slow code snippets or lines instead of the full code gives better results. So we followed that.

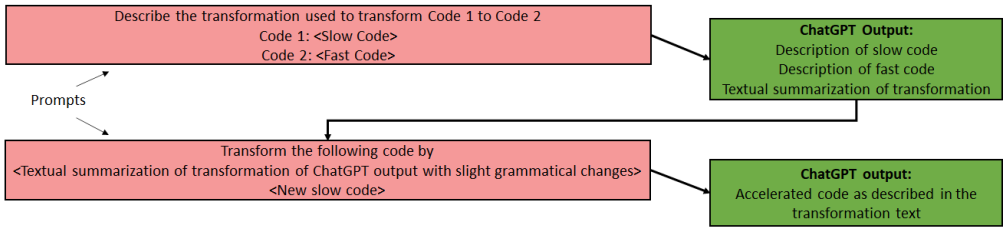


Figure 5: Schematic representation of Domain-Driven method

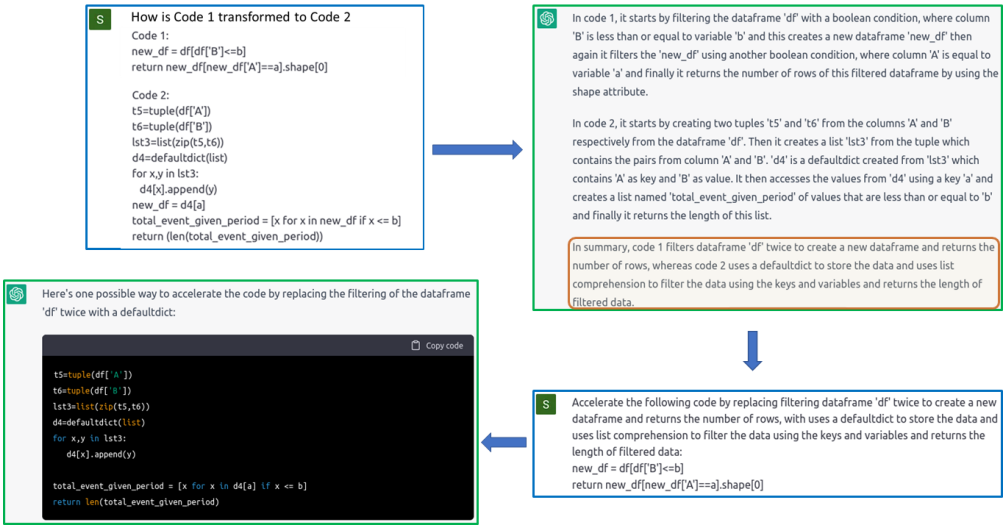


Figure 6: Sample Acceleration using Domain-Driven method

## 4 Empirical Observations and Discussion

We experimented on a Linux machine with 24 physical cores and 64 GB or RAM; however, our code used only a single core. We had test cases with sample data frames. Table 1 shows the empirical result. The first column denotes the Data-point index. The second column refers to the slow code snippet's time consumed (latency). The third column refers to the time consumed by the hand-accelerated code (gold data). The fourth column refers to the time the accelerated code obtained using the Domain-Driven method consumes. Columns 5 to 9 denote the time consumed by the codes generated by ChatGPT using the Generic-NL method.

In the Domain-Driven method, column 4, we can see that except for a single case (Data point 3, shown in red), the predicted code is faster than the handwritten code (19 out of 20 times). Moreover, for 11 out of 20 data points, the speedup is more than 2× (shown in blue). This is because it did the trick that was missed during hand acceleration. So this method is a huge success. But in this method, first, we would need to cluster the dataset into acceleration method clusters. Then, we need at least one data point from each cluster to be hand accelerated, which is overhead.

Table 1: Empirical evaluation results over 20 data points (slow code snippets)

| Data Point | Time taken in seconds to process a data frame with 500k rows | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Slow Code | Hand Accelerated | ChatGPT Generated Code | | | | | |
| | | | Domain-Driven method | Generic-NL Method (Top 5 results) | | | | |
| | | | | Best | $2^{nd}$ | $3^{rd}$ | $4^{th}$ | $5^{th}$ |
| 1 | 11.30 | 0.87 | 0.73 | 2.86 | 3.93 | 8.78 | 19.27 | 21.5 |
| 2 | 2.89 | 0.93 | 0.55 | 1.4 | 2.35 | 2.37 | 6.06 | 18.65 |
| 3 | 5.21 | 3.59 | 4.33 | 3.5 | 3.77 | 21.06 | - | - |
| 4 | 3.49 | 0.74 | 0.16 | 2.04 | 2.87 | 22.95 | 23.43 | 58.38 |
| 5 | 4.24 | 0.73 | 0.69 | 3.31 | 3.36 | 3.57 | 3.64 | 22.79 |
| 6 | 5.77 | 1 | 0.01 | 3.65 | 5 | 5.64 | 10.37 | 23.54 |
| 7 | 5.39 | 2.04 | 0.89 | 3.94 | 4.21 | 5.32 | 7.01 | 131.87 |
| 8 | 5 | 0.86 | 0.62 | 2.39 | 2.43 | 2.78 | 37.07 | 39.28 |
| 9 | 4.46 | 1.06 | 0.87 | 3 | 3.82 | 46.53 | 59.32 | 61.38 |
| 10 | 4.39 | 0.79 | 0.77 | 2.1 | 2.12 | 3.62 | 4.08 | 5.61 |
| 11 | 3.58 | 0.82 | 0.8 | 2.1 | 2.13 | 3.4 | 3.47 | 5.51 |
| 12 | 3.95 | 0.76 | 0.76 | 1.48 | 1.61 | 2.58 | 2.92 | 20.55 |
| 13 | 3.84 | 1.93 | 0.83 | 3.49 | 3.72 | 3.81 | 4.42 | 46.5 |
| 14 | 3.65 | 1.88 | 0.72 | 3.53 | 3.56 | 3.61 | 3.66 | 21.31 |
| 15 | 4.9 | 1.95 | 0.76 | 2.8 | 2.82 | 3.54 | 3.66 | 3.87 |
| 16 | 4.73 | 1.94 | 0.86 | 3.93 | 3.98 | 4.12 | 4.2 | 21.3 |
| 17 | 4.85 | 2.06 | 0.71 | 3.96 | 4.5 | 4.64 | 4.96 | 37.76 |
| 18 | 6.9 | 2.47 | 0.77 | 5.09 | 6.23 | 7.45 | 9.43 | 11.29 |
| 19 | 8.46 | 1.95 | 0.95 | 3.1 | 3.29 | 5.41 | 5.96 | 131.33 |
| 20 | 6.18 | 1.93 | 0.96 | 3.72 | 5.54 | 5.65 | 5.88 | 6.41 |
| 21 | 77.89 | 25.02 | _ | 22.5 | 25.4 | 27.45 | 28.58 | 29.52 |
| 22 | 120.49 | 67.07 | _ | 66.02 | 66.46 | 67.99 | 69.38 | _ |
| 23 | 67.82 | 0.048 | _ | 0.022 | 0.046 | 0.047 | 0.053 | 0.079 |
| 24 | 34.12 | 14.46 | _ | 13.46 | 14.01 | 14.51 | 14.65 | 14.93 |

The codes generated by the Generic-NL method (columns 5 to 9) are either slightly faster than the slow code or slower than the slow code. The timings mentioned in brown denote the codes generated by ChatGPT, which were faster than slow code. We only show the top 5 results from many versions generated using the Generic-NL method. Some of the generated codes were slower than the slow code because ChatGPT does not have the context of the data of the code it is operating on. However, for each data point, at least one version is significantly faster than the slow code. This is promising, as ChatGPT is directly predicting accelerated codes here. There are no more steps or human involvement required.

In the Generic-NL method, sometimes, we need to stop using the same chat and start with a new chat, to get better variability in the prediction from ChatGPT. Otherwise, after some time, ChatGPT is prone to repeat predictions very early. This may be because, even though ChatGPT has long context windows, code outputs eat up the context window fast so that ChatGPT can access just a couple of past conversation pairs. So, in the Generic-NL method, we surely need to start in a new chat for different data points. But also, while iterating for a data point, we may need to start a new chat to get better variability. This leads to an issue of deciding when to switch to a new chat and when to continue with the same chat. We started a new chat after we got repetitions for five consecutive prompts. And checked in the new chat if we get better variability.

For the Domain-Driven method, it is better to stay on the same chat while accelerating all codes of a code cluster and move to a new chat for the new cluster. Sometimes codes predicted by ChatGPT throw some errors. If we provide the error description and the final line from the stack trace that caused the error, ChatGPT can identify the cause and correct it in most cases. It, however, fails if the line of code is long and the error is something generic line syntax error. Sometimes ChatGPT suggests versions in the Generic-NL method that needs Major changes in the slow code, like using Cython or Numba compiler. Such predictions are generally mostly erroneous, making it difficult for ChatGPT to correct errors in such cases. So we have skipped such outputs in the Generic-NL method.

### 4.1 SOME INTERESTING OBSERVATIONS

ChatGPT itself is strong enough to identify not only the deadlines of code but code portions(sometimes part of a line) that do not affect the final output or return value. For example, our slow code had a sorting call to the data frame. This call was irrelevant as the actual order of the rows did not matter in the final return value. This was identified in the hand-accelerated version, and the sorting call was removed. ChatGPT was able to do the same without any intimation from our side. IT even notifies in its explanation that the sorting call was irrelevant and was thus removed. This shows the power of ChatGPT.

ChatGPT, along with the accelerated code, also describes the acceleration and why it thinks the out code may be faster than the slow code. It also does complain about the lack of enough context. We have tried to give hardware information, like whether we want our code accelerated for CPU or GPU in the Generic-NL method. ChatGPT can follow that and give predictions that work only on CPU or GPU.

## 5 RELATED WORK

**System level Acceleration:** There are some libraries for doing parallelization of Python code like Modin Zhang et al. (2021), Dask Rocklin (2015), and some schedulers across heterogeneous hardware like Weld Palkar et al. (2017a) Palkar et al. (2017b), Split Palkar & Zaharia (2019a) Palkar & Zaharia (2019b) from Stanford.

**Code Generation/Correction:** We tried a few methods to do code acceleration before using ChatGPT cgp. The method of using an IR-based template representation of the transformation is non-scalable and needs humans in the loop. Transformation of AST (Abstract Syntax Tree) of slow code to AST of fast Code using either CFG (Context Free Grammar) based or Grammar-based approaches would need to generate or create the CFG. GittaWinters & De Raedt (2020) that automates this is slow and not robust at all. Autopandas Bavishi et al. (2019) does not effectively explore the function search space and doesn't support conditional arguments. DL code correction method Recorder Zhu et al. (2021) needs lots of training data to train transformations.

**Description transformation:** Another method is to use a conversion pipeline-based solution, of slow code to a description, slow code description to a fast code description, and then from the fast code description, generate the fast code. CODEX Chen et al. (2021), and also even ChatGPT Castelvecchi (2022) can do text-to-code and code to text but we can't train a text-to-text transformation model with so little data. And in-context learning using such a large context is inefficient too.

We are the first to try to tackle the problem of automatic code acceleration, using LLM or otherwise.

## 6 FUTURE WORK

In Method One, our analysis showed that the predicted codes demonstrated only marginal performance improvement. To address this, our team is actively pursuing ways to enhance the accuracy of the code predictions.

As part of this effort, we plan to conduct a deeper exploration of Error Correction using ChatGPT and to test the system using various code correction datasets. In addition, we intend to provide ChatGPT with more data, such as the input data frame or test cases, and upgrade the underlying hardware and software configuration. This will help us achieve hardware and software configuration-based code generation, improving the system's performance.

It is important to note that the development environment often differs from the production environment, which can negatively impact the performance of the code. To mitigate this risk, our future work will include the development of performant transliteration techniques to ensure consistent performance as the infrastructure evolves.

## REFERENCES


Chatgpt. https://openai.com/blog/chatgpt/.

Chatgpt. https://huggingface.co/imvladikon/bert-base-uncased-jigsaw.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. Autopandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

Niklas Bjork and Charilaos Christopoulos. Transcoder architectures for video coding. *IEEE Transactions on Consumer Electronics*, 44(1):88–98, 1998.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

Davide Castelvecchi. Are chatgpt and alphacode going to replace programmers? *Nature*, 2022.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

Javier De la Rosa and Andrés Fernández. Zero-shot reading comprehension and reasoning for spanish with bertin gpt-j-6b. 2022.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pp. 990–998. PMLR, 2017.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Joseph M Hellerstein, Jeffrey Heer, and Sean Kandel. Self-service data preparation: Research to practice. *Data Engineering*, pp. 23, 2018.

Gabriel T Meyrer, Denis A Araújo, and Sandro J Rigo. Code autocomplete using transformers. In *Intelligent Systems: 10th Brazilian Conference, BRACIS 2021, Virtual Event, November 29–December 3, 2021, Proceedings, Part II*, pp. 211–222. Springer, 2021.

Mayank Mishra, Archisman Bhowmick, and Rekha Singhal. Fasca: Framework for automatic scalable acceleration of ml pipeline. In *2021 IEEE International Conference on Big Data (Big Data)*, pp. 1867–1876. IEEE, 2021.

Shoumik Palkar and Matei Zaharia. Optimizing data-intensive computations in existing libraries with split annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 291–305, 2019a.

Shoumik Palkar and Matei Zaharia. Optimizing data-intensive computations in existing libraries with split annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 291–305, New Yorlk, NY, USA, 2019b. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359652. URL https://doi.org/10.1145/3341301.3359652.

Shoumik Palkar, James Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. Weld: Rethinking the interface between data-intensive applications. *arXiv preprint arXiv:1709.06416*, 2017a.

Shoumik Palkar, James J. Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. Weld: Rethinking the interface between data-intensive applications. *CoRR*, abs/1709.06416, 2017b. URL http://arxiv.org/abs/1709.06416.

Luca Pasquini, Stefano Cristiani, Ramón García López, Martin Haehnelt, Michel Mayor, Jochen Liske, Antonio Manescau, Gerardo Avila, Hans Dekker, Olaf Iwert, et al. Codex. In *Ground-based and Airborne Instrumentation for Astronomy III*, volume 7735, pp. 957–968. SPIE, 2010.

Gil Press. Cleaning big data: Most time-consuming, least enjoyable data science task, survey says. https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says?sh=14bf3656f637.

Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 130, pp. 136. SciPy Austin, TX, 2015.

Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653*, 2023.

Ignacio G Terrizzano, Peter M Schwarz, Mary Roth, and John E Colino. Data wrangling: The challenging yourney from the wild to the lake. In *CIDR*. Asilomar, 2015.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Thomas Winters and Luc De Raedt. Discovering textual structures: Generative grammar induction using template trees. *Proceedings of the 11th International Conference on Computational Creativity*, pp. 177–180, 2020.

Andrew Zhang, Richard Lin, Sean Meng, and Crystal Jin. *Modin OpenMPI compute engine*. PhD thesis, Master's thesis, EECS Department, University of California, Berkeley, 2021.

Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 341–353, 2021.