Utilizing Large Language Models as no-code Interface in a Software Development Toolkit

Jukka Keisala

Bachelor's thesis
May 2023
Information and Communication Technology

Description

Keisala, Jukka

Utilizing Large Language Models as No-code Interface in a Software Development Toolkit

Jyväskylä: Jamk University of Applied Sciences, May 2023, 82 pages.

Degree Programme in Information and Communication Technology. Bachelor's thesis.

Permission for open access publication: Yes

Language of publication: English

Abstract

Large language models and their applications, such as ChatGPT and Bing AI, have gained much hype and visibility in media in past months (winter 2022–2023). Big companies like Microsoft, OpenAI, Meta and now Google have invested billions of dollars trying to keep themselves in pole position in the technology regarding artificial intelligence and machine learning.

The development of AI has been going on for years, but now the results and applications are available to almost everyone. The usage of large language models is free, with certain restrictions, and models can be used for different use-cases. OpenAI was the first company to monetize their models also for companies that are ready to pay extra to get faster responses and to utilize models in their applications and use-cases.

The assignment comes from the client company that asked to investigate the utilization of large language models to generate code that could be used in the analytics add-on of their IoT-based product.

In the thesis, the theory and methodology is related to large language models are presented, model catalog is presented and models are investigated, dataset is created, proper models are selected, test bench is created, fine-tuning is done, quantitative results are analyzed and conclusions are presented. The test framework, i.e., code-basis and dataset structure, was written during thesis writing and was designed in such way it could be used in further development of current project, but also in similar future projects.

Results achieved were encouraging and will be used in the product, at least as an experimental feature.

Keywords/tags (subjects)

Large language models, artificial intelligence, machine learning, transformers, software development kit.

Miscellaneous (Confidential information)

None

Kuvailulehti

Keisala, Jukka

Suurten kielimallien hyödyntäminen koodittoman rajapinnan avulla ohjelmistokehitystyökalussa

Jyväskylä: Jyväskylän Ammattikorkeakoulu, Toukokuu 2023, 82 sivua

Tekniikan ala. Tieto- ja viestintätekniikan tutkinto-ohjelma. Opinnäytetyö AMK.

Verkkojulkaisulupa myönnetty: kyllä

Julkaisun kieli: englanti

Tiivistelmä

Suuret kielimallit sekä niiden sovellutukset, kuten ChatGPT sekä Bing AI, ovat viime kuukausina saaneet osakseen paljon huomiota sekä palstatilaa eri medioissa (talvi 2022–2023). Suuret yritykset kuten Microsoft, OpenAI, Meta ja nyt myös Google ovat investoineet miljardeja dollareita pysyäkseen mukana tekoäly- ja koneoppimiskehityksessä.

Viime vuosina tekoäly on kehittynyt huimaa vauhtia, mutta vasta nyt tulokset sovellutuksineen ovat tavallisen käyttäjän saatavilla. Suurten kielimallien käyttö on tietyin ehdoin ilmaista, ja näitä malleja voidaan käyttää eri tarkoituksiin. OpenAI on ensimmäisenä yrityksenä alkanut tarjota mallejansa myös yrityskäyttöön tavalla, jossa malleja voidaan hyödyntää omaan bisnestarkoitukseen maksua vastaan.

Tehtävänanto tulee asiakasyritykseltä, joka on pyytänyt tutkimaan suurten kielimallien hyödyntämistä koodingenerointitehtäviin heidän IoT-tuotteen analytiikka-lisäosaan liittyen.

Tässä opinnäytetyössä esitellään suuriin kielimalleihin liittyvä teoria ja metodologia, esitellään ja tutkitaan saatavilla olevaa mallikatalogia, luodaan datasetti, valitaan sopivat mallit tarkempaan tutkimiseen, luodaan testipenkki, ”fine-tuunataan” (lisäkoulutetaan) mallia, ja analysoidaan malliajojen tuloksia ja tehdään johtopäätökset. Koodipohjasta sekä datasetistä koostuva testikehys suunniteltiin siten, että sitä voidaan hyödyntää sekä jatkokehityksessä että muissakin vastaavantyyppisissä projekteissa.

Saavutetut tulokset olivat rohkaisevia ja niiden pohjalta tehtyä sovelluslisäosaa tullaan käyttämään lopputuotteessa, ainakin niin sanottuna kokeellisena ominaisuutena.

Avainsanat (asiasanat)

Suuret kielimallit, tekoäly, koneoppiminen, transformerit, ohjemistokehitystyökalu.

Muut tiedot (salassa pidettävät liitteet)

Ei mitään

Contents

Abbreviations 8

1 Introduction 9

1.1 Background 9

1.2 The Client of the Thesis 10

1.3 IoT-TICKET - The Product Behind the Assignment 10

1.3.1 Product Description 10

1.3.2 DAS add-on to IoT-TICKET 12

1.4 The Assignment 13

1.5 The Objective of the Thesis 13

1.6 Research Questions 14

2 Large Language Models in the Light of AI Model Lifecycle 15

2.1 AI Model Lifecycle 15

2.1.1 Business Understanding 15

2.1.2 Data Understanding 15

2.1.3 Data Preparation 16

2.1.4 Modeling 17

2.1.5 Evaluation 18

2.1.6 Deployment 18

2.2 On Large Language Models 19

2.2.1 Definition of Large Language Model 19

2.2.2 Transformer Architecture 20

2.2.3 Generative Pretrained Transformers 22

2.3 Advanced Usage of Large Language Models 22

2.3.1 Prompt Engineering 22

2.3.2 Fine-Tuning 24

2.4 Model Endpoints 25

2.5 Tokens 26

3 Data Preparation and Model Selection 27

3.1 Dataset Description 27

3.1.1 Data Related to DAS Feature One 27

3.1.2 Data Related to DAS Feature Two 32

3.2 Dataset Structure and Content 38

3.2.1 Starting Point 38

3.2.2 Folder Structure 39

3.2.3 Data Samples 41
3.2.4 Division to Prompt-Engineering and Fine-Tuning 44
3.3 Available Model Catalog 45
3.3.1 OpenAI 46
3.3.2 Bing AI 47
3.3.3 Meta AI 48
3.3.4 Github Copilot 49
3.3.5 Other 49
3.4 Model Selection for Test Runs 50
3.5 Scoring 51
4 Test Runs and Findings 52
4.1 Test Bench 52
4.2 Runs 54
4.2.1 General Observations 54
4.2.2 GPT-3.5-turbo and GPT-4 55
4.2.3 code-davinci-002 55
4.2.4 Curie (Fine-Tuned) 56
4.2.5 Bing AI 60
4.3 Results 61
4.3.1 Feature 1 61
4.3.2 Feature 2a 64
4.3.3 Feature 2b 66
4.3.4 Summary 69
4.4 Code Implementation to SDK 70
5 Conclusions 73
5.1 Objectives and Results 73
5.2 Ideas for Future Development 74
References 75
Appendices 78
Appendix 1. Price Table of Test Runs 78
Appendix 2. Samples of Generated Code in Feature 1 79
Appendix 3. Sample of Generated Code in Feature 2a 80
Appendix 4. Feature 2b: Full Quantitative Distribution 81
Appendix 5. Samples of Generated Code in Feature 2b 82

Figures

Figure 1. Transformer model as a black box running machine translation task. (Alammar, 2018)20
Figure 2. Transformer contains encoders half and decoders half. (Alammar, 2018)................20
Figure 3. Encoder and decoder stacks. (Alammar, 2018)..21
Figure 4. Encoder and decoder are formed of layers. (Alammar, 2018)..................................21
Figure 5. Few-Shot learning example (adapted from Brown et al., 2020). 23
Figure 6. One-Shot learning example (adapted from Brown et al., 2020). 23
Figure 7. Zero-Shot learning example (adapted from Brown et al., 2020)...............................24
Figure 8. Fine-tuning example (adapted from Brown at al., 2020). 24
Figure 9. Time series forecast, example plot. 28
Figure 10. Databricks notebook job registration code to IoT-TICKET.......................................28
Figure 11. Registered model in IoT-TICKET...29
Figure 12. Request payload to Databricks jobs API. 29
Figure 13. High-level picture of utilizing LLM to DAS..30
Figure 14. Example Python code as system input. 31
Figure 15. Example LLM output. 32
Figure 16. Expected JSON payload for example function...34
Figure 17. Response payload after function calculation. 34
Figure 18. Schemas before registering function to IoT-TICKET. 35
Figure 19. Example of supposed generated output from LLM using data for feature 2a.........36
Figure 20. Generated Python function implementation related to dataset of feature 2b. 38
Figure 21. Dataset folder structure...39
Figure 22. Sample of DAS feature 1 context file..41
Figure 23. Sample of test file of DAS feature 1..42
Figure 24. Sample file from a feature2a/test folder. 42
Figure 25. User-prompt and label from feature2b/test folder file..43
Figure 26. Test payload from feature2b/test folder file..44
Figure 27. JSONL formatted training data example...45
Figure 28. Dataset read in test bench. 52
Figure 29. GPT-3.5-turbo model access through public API using OpenAI python library.........53
Figure 30. Example of RunResult for one run (transposed for more compact view). 54
Figure 31. Fine-tuning of Curie model is on running phase in Azure OpenAI service. 57
Figure 32. Training and validation losses from curie model fine-tuning. 58
Figure 33. Test run on das-curie model in Azure OpenAI playground. 59

Figure 34. Feature 1: Sums of models’ scores per run and system-prompt. 62
Figure 35. Feature 1: Quantitative distribution of score values across test set. 63
Figure 36. Feature 2a: Sums of models’ scores per run and system-prompt. 64
Figure 37. Feature 2a: Quantitative distribution of score values across test set. 65
Figure 38. Feature 2b: Sums of models’ scores per run and system-prompt. 66
Figure 39. Feature 2b: Best context selection after aggregations. Uses only steerings 1 and 3. 67
Figure 40. Feature 2b: Partial quantitative distribution of score values across test set. 68
Figure 41. Asking generation from command line. 72

Tables

Table 1. Size of the dataset in number of files. 40
Table 2. Parameter combinations for code-davinci-002 runs. 56

Abbreviations

AI	Artificial Intelligence. Refers to the development of computer systems that can perform tasks that would typically require human intelligence such as learning, problem-solving and decision-making.
CRISP-DM	Cross-Industry Standard Process for Data Mining. Widely used methodology for developing data mining projects.
DAS	Data Analytics Suite. Refers to the add-on to IoT-TICKET product (by Wapice Ltd). Brings AI and ML capabilities on top of the IoT applications.
DL	Deep Learning. A subset of machine learning that involves training deep neural networks with multiple layers to learn representations of data.
GPT	Generative Pretrained Transformer. Refers to a family of large language models developed by OpenAI for natural language processing tasks.
IoT	Internet of Things. Refers to the network of physical objects connected to the internet that can collect and exchange data.
ML	Machine Learning. Subset of AI that involves training algorithms to learn patterns and make predictions or decisions based on data.
MLOps	Machine Learning Operations. A practice based on DevOps principle that makes machine learning workflows and model management automated and streamlined.
NN	Neural Networks. A type of machine learning algorithm that models the structure and function of the human brain.
ONNX	Open Neural Network eXchange. An open-source format for representing deep learning models that allows interoperability between different deep learning frameworks.

1 Introduction

1.1 Background

Artificial intelligence is a hot topic today. Together with its subsets machine learning, neural networks and deep learning, artificial intelligence is such a hot topic that you may face the news concerning AI in your everyday life.

Most of the news concerns applications of AI, whether they are good or not. Behind the applications there is theory and practice. In theory level, there is ongoing research on methods and methodologies. On the practice level, the computing power is greater than ever and the data available is huge. Putting theory and practice together, we see constant evolution in this area and the applications come better and better and closer to the end-user.

Last months (at the time of writing, March 2023) hot news are related to generative AI models, such as GPT-3, Codex and Dall-E from OpenAI company and AI-powered search engine Bing by Microsoft. These models have gained huge popularity in the news but also in “scene”. These generative models primarily produce text or images from user input.

Some people see threats that AI causes, some opportunities. At this point the hot-topic models are good, let me say very good, but not almighty. It is quite easy to fool the model and make it look embarrassing. Therefore, it is tempting to say that those are bad models. It is not reasonable to judge the models so easily, it is more interesting to find applications where they are good at.

In software companies and among the community of software developers and data scientists etc., there is one use-case that interests us more than the others: how well those models can generate code? In addition, if the models can generate code with known coding languages, it is possible that these models could be utilized to generate code with non-seen language!

The topic of this thesis is closely related to this question setting. We want to find out if and how the models can be used to generate code or commands that are certainly not in the training set of the models, since those are private code that the client company owns and not available on the internet.

1.2 The Client of the Thesis

The client of this thesis is Wapice Ltd, a Finnish full-service software company established in 1999. In addition to the subcontracting services that Wapice Ltd provides, it has its own products and services that are being used by domain leading industrial companies around the world. (Wapice website n.d.)

Wapice products include IoT-TICKET® Nordic IoT and AI platform, Summium CPQ sales configurator tool, Summium Selector web shop platform, EcoReaction energy reporting service, WRM247 edge device for IoT and CANrunner analyzer & diagnostic tool. (ibid)

Service offerings include AI and data science services, cloud services, consulting, DevOps, electronics design and device manufacturing, security services, IoT services, web and mobile services and so on. (ibid)

The topic of this thesis belongs partly to IoT-TICKET product and AI & data science services. More closely, the results of the thesis are expected to give added value to IoT-TICKET customers by making analytics and AI workload deployment easier and faster, and on the other hand, the know-how related to large language models and related topics gives more sales opportunities and raises the profile of Wapice.

1.3 IoT-TICKET - The Product Behind the Assignment

1.3.1 Product Description

IoT-TICKET® is an innovative, subscription based IoT and AI platform, designed for creating market ready solutions within minutes and without a single line of code. It is marketed to be the most advanced IoT software and service, and it scales from specific use cases to massive scale projects, thus servicing customers from medium sized companies to global corporations. (IoT-TICKET website, n.d.)

IoT-TICKET® offers major cost savings and faster time-to-market. It is offered as monthly subscription model from secure IoT-TICKET® cloud or deployed as on-premises installation to customer premises, and it integrates directly into existing ecosystems. As a major acknowledgement, it won the Microsoft Global Application Innovation Award in 2019. (ibid)

IoT-TICKET's main application domains are industry, smart cities, property maintenance, moving machines, smart logistics and smart manufacturing. IoT-TICKET's website holds information for many more customer cases but here are three example cases shortened and summarized by the author from the original customer stories.

Case 1: "Danfoss Drives uses IoT and AI to provide top-class lifecycle solutions". Danfoss monitors frequency converters (used to control frequency and voltage of an electric motor's power supply) using different sensors and transfer measurements to IoT-TICKET. Using IoT-TICKET and its features Danfoss can monitor the converters in real-time and thus gain benefits like cost savings in maintenance, rapid response to faults through immediate alerts and higher availability. Utilizing AI Danfoss can do life cycle predictions to key components.

Case 2: "Schaeffler selects Wapice IoT-TICKET as its global IoT platform". For advanced condition monitoring, Schaeffler launched a new solution that utilizes battery powered, wireless mesh network sensors connected to the cloud. Using IoT-TICKET as a backbone, Schaeffler can easily and quickly create massive-scale, production-scale IoT applications.

Case 3: "Helen chose Wapice Energy Services to monitor solar power plants". Solar power plants need to be monitored and serviced regularly to ensure their safe and efficient operation. Real-time monitoring provides peace and security for solar power plant owners. The solution automatically generates a forecast of future solar power production. The monitoring solution also helps Helen offer its customers predictive maintenance and efficient repair of latent faults in solar power systems.

As its core, IoT-TICKET is a low-code/node-code platform, meaning one does not need to have programming skills to use it. Different data visualization, data logic and management views can be created by combining ready-made components using the tools provided by the platform. This allows experts from different industries to directly apply their own expertise to create solutions without having to have a long and complex software development project in between the idea and the finished solution. (IoT-TICKET website, n.d.)

1.3.2 Data Analytics Enhancements for IoT-TICKET

There was identified customer needs for more demanding analytics capabilities for IoT-TICKET, and to answer this need, more advanced analytics features have been developed. The result is a feature set called DAS (Data Analytics Suite). It offers different tools and ways to leverage several machine learning and AI platforms and solutions to work seamlessly within IoT-TICKET. The first two features of DAS are treated in this thesis and will be investigated more closely next.

The **first** feature is the integration between IoT-TICKET and Databricks (a Data Lakehouse Architecture and AI company, whose product name is the same as the company name). The following quote (Introduction page at Databricks website, n.d.) explains shortly what Databricks is:

"The Databricks Lakehouse Platform provides a unified set of tools for building, deploying, sharing, and maintaining enterprise-grade data solutions at scale. Databricks integrates with cloud storage and security in your cloud account and manages and deploys cloud infrastructure on your behalf."

Databricks also provides an environment familiar to data scientists using Python: IPython-style notebooks. Data scientists can write their code into these notebooks and parametrize the notebooks so that the parameter values can be given in the event of running the notebook. (ibid).

The integration and the interoperability between IoT-TICKET and Databricks is based on the above: the interesting data from IoT-TICKET is send to Databricks' notebooks using the notebook parametrization, calculation is done notebooks, and then the results are sent back to IoT-TICKET, if wanted. Due to the nature of notebooks, the procedure is asynchronous. To make the interoperability work, Wapice has developed a developer toolkit to solve the problems. That toolkit is called **DAS-SDK**, Data Analytics Suite Software Development Toolkit, made by Python. The abilities of DAS-SDK are considered later.

The **second** feature of DAS is the integration between IoT-TICKET and (supported) HTTP services. The idea is that the IoT-TICKET access HTTP-services' endpoints using IoT data in request payload, the service does the calculation and then returns the results in the response payload. The procedure is synchronous in its nature and so results can be consumed in IoT-TICKET right away when the result is returned.

To get things started, the first supported HTTP services are Azure Function-app (a serverless service to run functions) with HTTP-triggering and Databricks model serving (trained model is exposed behind an HTTP endpoint). In theory, any HTTP service can be accessed but to make the interoperability easy, some helper tools are needed. In DAS-SDK, Wapice provides tools to make interoperability between IoT-TICKET and Azure Function-app & Databricks model serving easy.

In addition to the data analytics features, DAS-SDK also has a client library for IoT-TICKET REST API so users can create, read, update and delete resources available there. DAS-SDK has been developed in the data scientists point-of-view, which means that the data they fetch from IoT-TICKET is automatically transformed into well-known formats like Pandas Dataframes and Numpy Arrays. Also, the tools for interoperability are done by modelling the objects in IoT-TICKET, Databricks and Azure Function-app using Python classes.

On top of Python, there is also a command-line interface available so selected workflows can be started from the command line. It has certain use-cases, mostly in quick queries without the need to go to Python interpreter and do importing and object instantiations.

1.4 The Assignment

The deputy of Wapice Ltd assigned the author to investigate how large language models could be used to make the usage of DAS features **one** and **two** easier and faster when using DAS-SDK. The assignment involved both the **theory part**, which is fine-tuning the model and/or prompt-engineering to get desired output, and the **implementation part**, that means implementing the solution into DAS-SDK.

1.5 The Objective of the Thesis

The main objective of this thesis is to answer the theory part mentioned above. This means that we need to investigate available models that could solve our problem and select the best of those.

These selected models will be accessed by a certain test dataset, which contains the input data for the model and the result data which will be compared to the outputs of the model. If the result data is close to the output of the model, the model is considered as potential. Potential models will be looked at more closely, and we use fine-tuning and prompt-engineering methods to see if the model accuracy could be enhanced.

The secondary objective is the implementation part. After finding best model and/or correct inputs to get the best results, we write an example code that adds an abstraction layer between the DAS-SDK tools and the model API so that the user does not have to know or care about the details and the user can concentrate on the usage itself.

The secondary objective is not so interesting in nature since it is just normal software development. Therefore, we put most of our effort into the main objective.

1.6 Research Questions

The objectives of this thesis were presented in the previous section. To keep the thesis under control and the focus on correct things, we declare the research questions shortly as follows:

1. Can large language models be utilized to generate non-seen code accurately enough?
2. Which models from model catalog are appropriate for question 1?
3. Is it better to use fine-tuning, prompt-engineering, or both?
4. How to implement the code into a software development kit to achieve a nice no-code interface for the user?

2 Large Language Models in the Light of AI Model Lifecycle

2.1 AI Model Lifecycle

Cross-industry standard process for data mining, short CRISP-DM, is an open standard process model that describes common approaches to data mining projects. It is “de-facto standard and industry-independent model for applying data mining projects” (Schröer et al, 2021). An article by Wirth and Hipp (2000) presents CRISP-DM methodologies compactly. It is not a research paper, but it nicely summarizes the idea and introduces the related references. Here the term “data mining” can be equally replaced by machine learning or artificial intelligence. CRISP-DM has six sequential phases which we introduce next.

2.1.1 Business Understanding

Like any good project, AI projects also start with deep understanding of what customers want. Understanding the business case helps to focus on the objectives and requirements of the project. Then this knowledge can be converted into an exact AI problem definition and project plan can be created to achieve these objectives.

In this thesis, the business case was quite clearly stated by the client. The client wanted the user experience of DAS features **one** and **two** should be such that there is no fear that the customer feels that features are too hard to use. If the feature is hard to use, it easily means that it is not used at all and at some point, the customer won’t be the customer anymore.

On the other hand, the client stated that the know-how with respect to large language models and their utilization is important for future business cases and projects.

2.1.2 Data Understanding

In the data understanding phase, a closer look into data is taken. Wirth and Hipp (2000) puts it as follows:

"The data understanding phase starts with an initial data collection and proceeds with activities in order to get familiar with the data, to identify data quality problems, to discover first insights into the data, or to detect interesting subsets to form hypotheses for hidden information."

This phase has four steps. Those are:

1. **Collection of initial data:** Data is acquired and loaded into analysis tool.
2. **Describing the data:** Data is examined and its properties (e.g., format and number of records) are documented.
3. **Data exploration:** Data is explored more closely, queries are made, visualizing data and preliminary relationships are noticed.
4. **Data quality verification:** Data quality is documented.

It would be good if every expert working on a data mining project took a short sight on raw data at some point. It helps to understand the business case better and to notice possible lacks in preprocessed (prepared) data.

2.1.3 Data Preparation

Data preparation is a phase where the final dataset is constructed from the original raw data. The final dataset here means that the data is ready to be passed into modeling tools like model training. (Wirth and Hipp, 2000)

It is likely that data preparation tasks are performed multiple times without any prescribed order (idis). The author confirms that based on his experience; there is no way to be sure beforehand what kind of attributes and features are needed in the modeling phase (next phase). Usually, knowledge from modeling phase helps to understand the problem and new ideas what dataset should include, arise.

The following steps are included in this phase:

1. **Selecting data:** Select which data will be included and excluded and rationale the decision.
2. **Cleaning data:** Data cleaning means correcting, imputing or removing erroneous values.
3. **Constructing data:** Deriving new attributes (like body mass index from height and weight).
4. **Integrating data:** Creating new datasets by combining data from multiple sources.

5. **Formatting data:** Actions like converting string-type numbers into numeric values so that mathematical operation can be performed.

2.1.4 Modeling

The modeling phase includes selecting and applying various modeling techniques as well as parameter calibration to optimal values. Usually there are several techniques that can be tried for the same type of AI problem.

This phase often is connected to the data preparation phase, since the prepared data is now used and if there are problems in the data, it becomes visible and repair actions are needed. On the other hand, one easily gets new ideas when trying different techniques and the need for a new dataset might arise.

Here are the steps:

1. **Selecting modeling techniques:** Determine which techniques and algorithms will be tried (e.g., neural network, classification, regression).
2. **Generating test design:** Splitting dataset into training, validation and test sets.
3. **Building model:** Model is trained.
4. **Assessing model:** Trained models are competing with each other. Model results are being interpreted using domain knowledge and pre-defined criteria. Better results do not necessarily mean a better model. This step is technical in its nature since the comparison is based on performance metrics that can be evaluated numerically.

Related to step 2, training data is used for model training and the model weights are based on training data. Validation data is used for hyperparameter optimizing process; then model weights are not changed but parameters of the model, hyperparameters, are being changed to find best ones. Test data is used after model training and possible hyperparameter optimization to see what the final performance of the model is. Test data is “hold out” data and should not be used in training.

2.1.5 Evaluation

At this phase one has built at least one model that has high quality in the data analytics point-of-view. One could be tempted to select the best performing model as the final model, but this should not be done without further hesitation. We need to be sure that the selected final model should meet the business objectives.

Evaluation phase steps are:

1. **Evaluating results:** Which models meet the business criteria?
2. **Reviewing process:** Review the work so far. Was the dataset biased? Was there enough data? Were all phases and steps properly executed?
3. **Determining next steps:** Which model goes to deployment? Which model is of further interest? Any ideas for new projects?

2.1.6 Deployment

Depending on the requirements for the project, deployment phase can mean many things. It might be for example that a report will be written on the project, a model is deployed into production, or the handover is done to the customer.

Although this is the final phase of the AI model lifecycle, it does not have to mean that the work ends. There are several phases that can be applied after deployment, such are model monitoring, fine-tuning and things like that. But these operations usually sit under term MLOps which is a method or family of methods for machine learning model lifecycle management that is usually automated. It is like DevOps for AI models.

The steps of this final stage are:

1. **Planning deployment:** Develop deployment plan and document it.
2. **Planning monitoring and maintenance:** Monitoring and maintenance plans should be made to avoid problems in the model's operational phase.
3. **Producing final report:** Final report and presentation should be made, also documenting lessons learned and sharing those.
4. **Reviewing project:** Project retrospective which gives possibility to see what went well, what could have been done better and ideas in the future.

2.2 On Large Language Models

2.2.1 Definition of Large Language Model

Large language models, or LLM for short, are deep learning algorithms that can recognize, summarize, translate, predict, and generate text and other content based on knowledge gained from massive datasets. (NVIDIA blog post, 2023)

They are based on models with **transformer architecture**, which are neural networks that learn context and thus the meaning. This is done by tracking relationships in sequential data, like words in a sentence. Moreover, *"transformer models apply an evolving set of mathematical techniques, called attention or self-attention, to detect subtle ways even distant data elements in a series influence and depend on each other." (ibid)*

As the name suggests, the word "large" comes from the fact that the dataset that the model was trained on is large. For example, the famous GPT-3 model (Generative Pretrained Transformer 3) has been trained using about 45TB of text data from different sources (Cooper, 2021) and the model requires 800GB to store (Wikipedia article). To be precise, GPT-3 model is actually a family of models of different sizes, as is the case for quite a lot of AI models. Depending on the resources available and the requirements for the model, different sizes can be selected for different cases to optimize the costs and performance.

2.2.2 Transformer Architecture

The first scientific paper that introduced transformers was “Attention Is All You Need” by Vasvani et al. (2017). Looking at the high level, transformer takes a sentence in and outputs the transformed sentence (Figure 1).

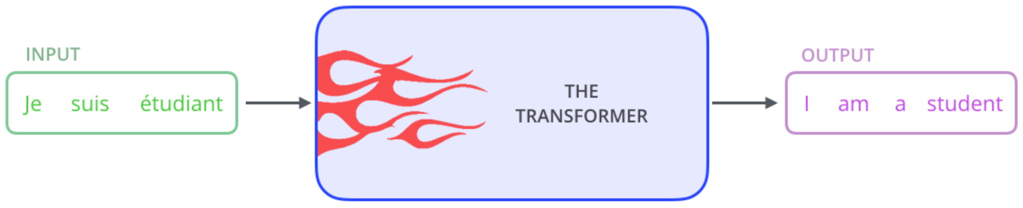


Figure 1. Transformer model as a black box running machine translation task. (Alammar, 2018)

By watching closer into transformer, there are two components: an encoding component and decoding component, see Figure 2. In general, an encoder is a model component that transforms input data into a lower-dimensional representation and decoder transforms lower-dimensional input back into original data format.

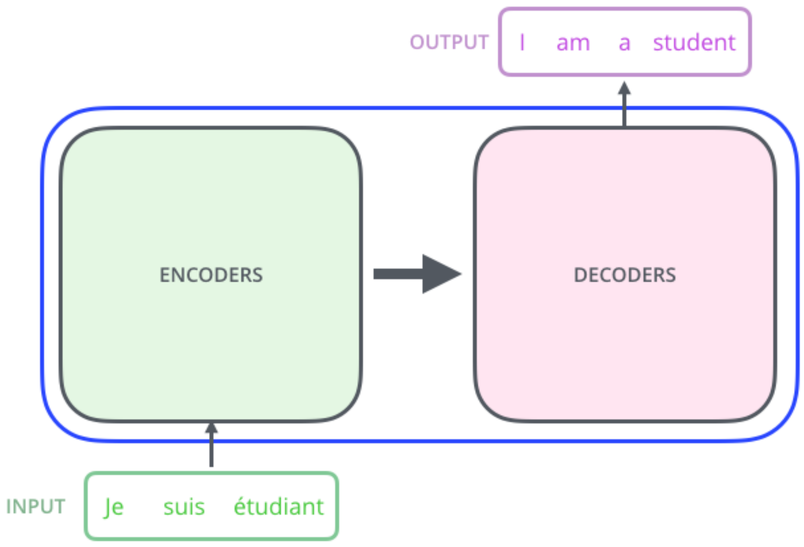


Figure 2. Transformer contains encoders half and decoders half. (Alammar, 2018)

Both components are in fact a stack of encoders and decoders (Figure 3). Pay attention that the data flows though encoders one after one but on the decoder side the situation is somewhat different. The features are available for every decoder in addition to the output of the previous decoder (except the first one).

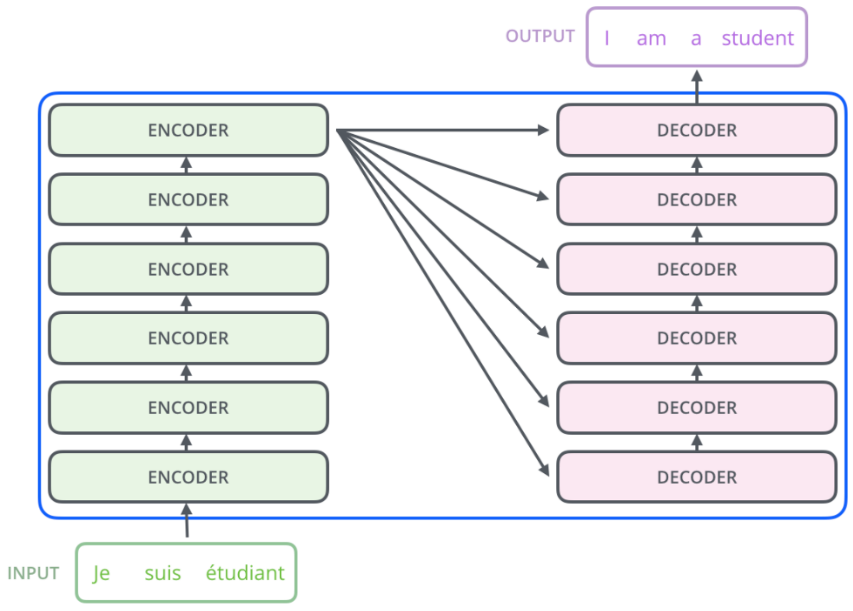


Figure 3. Encoder and decoder stacks. (Alammar, 2018)

Furthermore, encoders and decoders can be extracted as in Figure 4. Encoder has two layers – feed forward and self-attention, where decoder has one extra layer between these two, so-called attention layer.

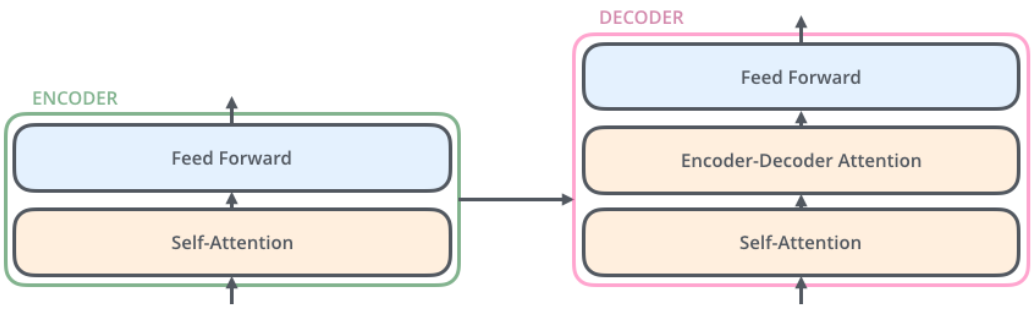


Figure 4. Encoder and decoder are formed of layers. (Alammar, 2018)

To summarize it, encoder(s) basically extracts features from an input sentence and the decoder(s) uses the features to produce an output sentence. We do not want to go too deep into the encoders and decoders. If the reader is interested in this topic, please see great article by Kikaben (2021) more closely.

2.2.3 Generative Pretrained Transformers

As one could guess from the naming, generative pretrained transformers are deep learning models that are of transformer architecture, pretrained and generative. Here generative simply means that it can generate something, usually sentences, code, images or sound.

GPT model family uses mainly the decoder half of the transformer architecture, meaning that they are focused on taking in embeddings (or features) and producing text. On the other hand, language models like BERT use the encoder half to generate embeddings (or features) from raw text which can be used in different machine learning applications. (Cooper, 2021)

Large language models are nowadays transformers. Transformers have taken place from recurrent neural networks and convolutional neural networks. (Kikaben, 2021)

Language models have many applications including machine translation, text classification, speech recognition, information retrieval, news article generation, question answering etc. (Cooper, 2021).

2.3 Advanced Usage of Large Language Models

2.3.1 Prompt Engineering

Prompt engineering, also known as in-context prompting, can be defined as follows:

"*A prompt is a set of instructions provided to an LLM that programs the LLM by customizing it and/or enhancing or refining its capabilities.*" (White et al, 2023).

In another words, it is about steering LLM behavior for desired outcomes without updating the model weights. To steer the model, there are three common learning methods for that (Brown et al.): few-shot, one-shot and zero-shot learning.

Few-Shot Learning

The term few-shot learning is used when the model is given few demonstrations of the task at inference time (idis). The example found at (idis) explains this nicely (Figure 5).

```
Translate English to French:     <-- task description
sea otter => loutre de mer       <-- example
peppermint => menthe poivrée     <-- example
plush girafe => girafe peluche   <-- example
cheese =>                        <-- prompt
```

Figure 5. Few-Shot learning example (adapted from Brown et al., 2020).

One-Shot Learning

The term one-shot learning is the same as few-shot learning but only one demonstration is allowed (idis). Here is the corresponding example (Figure 6).

```
Translate English to French:     <-- task description
sea otter => loutre de mer       <-- example
cheese =>                        <-- prompt
```

Figure 6. One-Shot learning example (adapted from Brown et al., 2020).

Zero-Shot Learning

Zero-shot learning is the same as one-shot except that no demonstrations are allowed, and only description of the task is given (idis). Then the corresponding example becomes to (Figure 7):

```
Translate English to French:     <-- task description
cheese =>                        <-- prompt
```

Figure 7. Zero-Shot learning example (adapted from Brown et al., 2020).

Other Learning Methods

In addition to learning methods presented above, there exists also the following learning methods as well: **Chain-of-Thought (CoT) Prompting**, **Zero-Shot CoT**, **Self-Consistency, Generated Knowledge Prompting** and **Automatic Prompt Engineering**. We do not go into details of these, but the interested reader should take a short tour on those by following prompt engineering guide (dair-ai Github page).

2.3.2 Fine-Tuning

Fine-tuning is a method where the weights of a pre-trained model is updated by re-training the model using supervised dataset that is specific to a certain task (Brown et al.). Here supervised dataset means a dataset that has numerous labeled examples (input and desired output/label). Usually, thousands or hundreds of thousands labeled examples are used in fine-tuning (idis).

Here is a corresponding translation example that can be contrasted to n-shot learning methods above (Figure 8).

```
sea otter => loutre de mer       <-- labeled example
          ⇓
    gradient update
          ⇓
peppermint => menthe poivrée     <-- labeled example
          ⇓
    gradient update
          ⇓
plush giraffe => girafe peluche  <-- labeled example
          ⇓
    gradient update
          ⇓
cheese =>                        <-- prompt
```

Figure 8. Fine-tuning example (adapted from Brown at al., 2020).

In Figure 8, the term gradient update means the same as model update, but the previous term is widely used in literature.

Prompt engineering and fine-tuning are not exclusive methods, so one can still use prompt engineering for fine-tuned models. But generally, fine-tuning is considered as an option when prompt-engineering fails to give results good enough.

2.4 Model Endpoints

Large language models generally have multiple endpoints. This means that a model can be used for different purposes or different ways. For example, OpenAI models have endpoints for completions, chat-completions, edits, moderation, translations etc. (OpenAI documentation page for models, n.d.). Quite popular among the popular applications is the chat-completion endpoint, in which the chat history (that is tied to roles and multi-turn conversations), can be given as the input and the model gives its answer based on the context defined by history. Therefore, that endpoint is used widely in chatbot-applications. Plain completion endpoint takes in just one prompt and the model will return the predicted completion. (idis)

The naming *completion* or *chat-completion* is not standard, but however widely used in the scene. In the common language, both refer to the process of generating text that completes a given prompt or input. For example, Mehdi (2023) uses the wordings “deliver better search, more complete answers, a new chat experience and the ability to generate content” without mentioning the word *completion*. However, it is easy to see what this is all about.

The use-case in this thesis is a bit of both, completion and chat-completion. The purpose is not to offer a possibility to chat with AI but instead, to get one and good enough response from AI. This objective can be achieved by combining pre-defined prompt with the user-prompt. For the user, it seems that the usage is a zero-shot method but in the backend, it is either chat-completion or completion, depending on the model and its performance.

2.5 Tokens

Tokens are basic units of text and code that a large language model can understand and process. The following quote give compact explanation what a token is:

"Tokens can be characters, words, subwords, or other segments of text or code, depending on the chosen tokenization method or scheme. Tokens are assigned numerical values or identifiers, and are arranged in sequences or vectors, and are fed into or outputted from the model." (Microsoft Learn documentation page, n.d.)

There are many tokenization methods, like rule-based, statistical and neural (ibid). Such method, or tokenizer, takes in piece of text and maps it to a number vector. The number vector is given as input to the model and the output is also a vector, that is transformed into text using the same tokenizer, but other way around (so called detokenization).

For example, if the sentence "I am a happy person" should be translated into Finnish, the tokenization process could be the following: "I" maps to 10, "am" maps to 20, "a happy" maps to 30 and "person" maps to 40, so the tokenization would be [10, 20, 30, 40]. Then this vector is sent to LLM, and the output could be [39, 212] which becomes, after detokenization, to "olen" "onnellinen".

It is known that tokenizers are not normally injective in the mathematical sense, which means that words like "happy" and "happily" might get tokenized into same (representative) value. This means that little information is lost in the tokenization process. Moreover, in the detokenization process, the representative text is selected. Therefore, some post-processing steps could be needed to get the resulting text grammatically correct.

3 Data Preparation and Model Selection

3.1 Dataset Description

The dataset that we consider is related to the DAS features **one** and **two** discussed in introduction. Both features are different in nature but similar in how they work. The first feature was designed to trigger Databricks jobs with IoT-TICKET data and parameters and the second feature was designed to call HTTP functions. They have different use-cases, but both work in that way that IoT-TICKET is programmed to send certain type of HTTP requests with certain payload.

We next present what kind of data we want to generate and on what basis.

3.1.1 Data Related to DAS Feature One

The workflow related to DAS feature one is the following:

- Design and implement the notebook.
    - Define what data, references and/or parameters are needed from IoT-TICKET.
- Deploy notebook to Databricks.
- Register notebook to IoT-TICKET.
    - Give instructions on how IoT-TICKET should run it.
- Run notebook block in dataflow.

For better understanding, we follow a simple example case. Let’s say we want to forecast new data points to given time series data (see Figure 9).

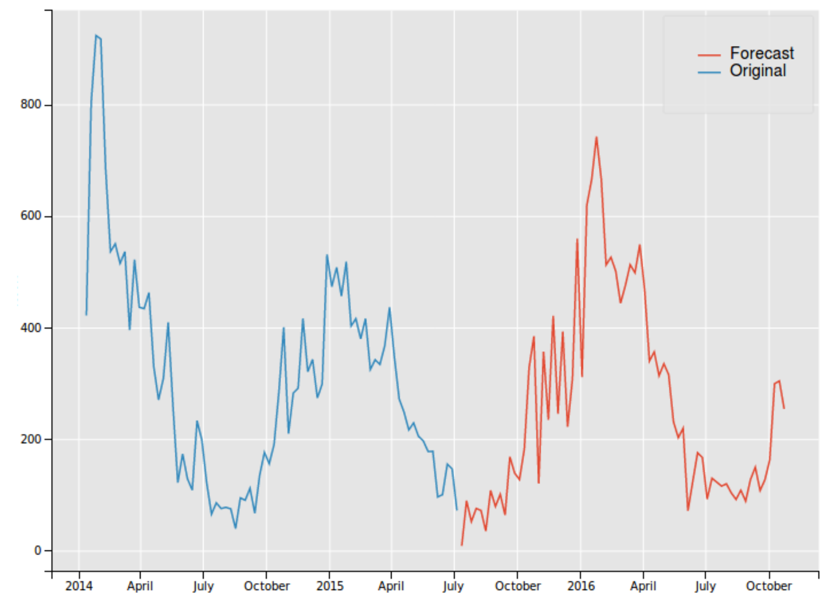


Figure 9. Time series forecast, example plot.

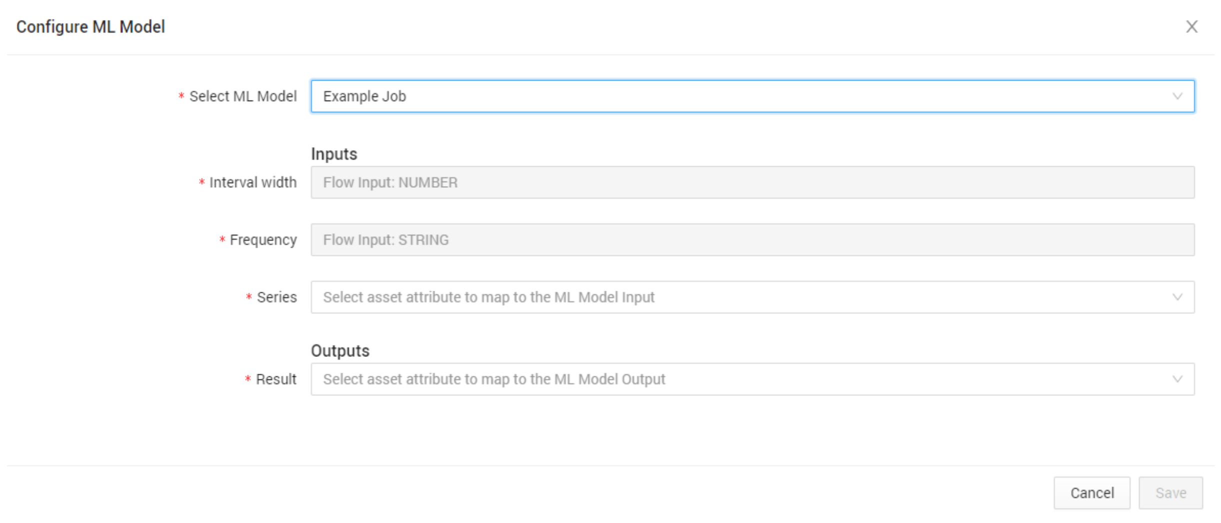
To make such a forecast, we need the following information from IoT-TICKET: time series data itself, number of how many steps forward the forecasting should be done, and the frequency of those steps. For example, if we get data for two years and want to forecast the next two months, we set steps=2 and freq="month" to get two new data points as a result.

To register this model into IoT-TICKET, that is to make IoT-TICKET know such model is available, we do it by using tools from DAS-SDK (see Figure 10).

```
from das import DatabricksJob, IoTClient
from das.helpers.schemas import NotebookNode

job = DatabricksJob(...)  # No interested in this

job.notebook_inputs =  [
    NotebookNode('Series', 'attribute_id', 'series_attribute_id'),
    NotebookNode('Steps', 'number', 'steps'),
    NotebookNode('Frequency', 'string', 'freq')
]
job.notebook_outputs = [
    NotebookNode('Result', 'attribute_id', 'result_attribute_id'),
]
iot = IoTClient()
iot.register_model(job.as_dict())
```

Figure 10. Databricks notebook job registration code to IoT-TICKET.

Next figure shows how the registered model looks like in IoT-TICKET (Figure 11).


Configure ML Model

* Select ML Model	Example Job
	Inputs
* Interval width	Flow Input: NUMBER
* Frequency	Flow Input: STRING
* Series	Select asset attribute to map to the ML Model Input
	Outputs
* Result	Select asset attribute to map to the ML Model Output

Figure 11. Registered model in IoT-TICKET.

After the registered model is configured, it is ready to be run. At this point we want to point out the naming convention to avoid hassle: registered model in IoT-TICKET is a representation of the notebook in Databricks. Notebook itself can run some real model or do something else, in our case it uses forecasting algorithm. When IoT-TICKET model is triggered, it sends an HTTP call to Databricks jobs API with payload shown in Figure 12.

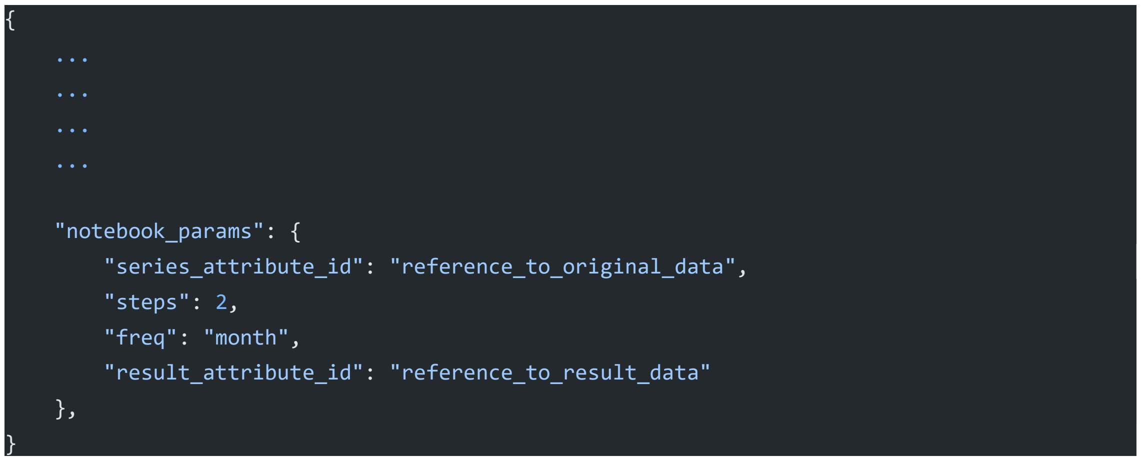

```
{
    ...
    ...
    ...
    ...

    "notebook_params": {
        "series_attribute_id": "reference_to_original_data",
        "steps": 2,
        "freq": "month",
        "result_attribute_id": "reference_to_result_data"
    },
}
```

Figure 12. Request payload to Databricks jobs API.

Here “series_attribute_id” is a reference where the original data can be fetched. Fetching happens in notebook. One could also pass raw data directly, but we are not doing that in this example. Also “result_attribute_id” is a reference to a point where the forecasted data should be written to by notebook.

What We Are After?

Now we have presented closely how DAS feature one is expected to be used. The process is somewhat simple in its steps and automated except the step where user needs to register the model to IoT-TICKET. User needs to know how DatabricksJob-class should be used. The most difficult part is to define the job inputs and outputs. This is the place where we want the help from large language models.

In Figure 13 we have an overview related to LLM and its inputs and outputs.

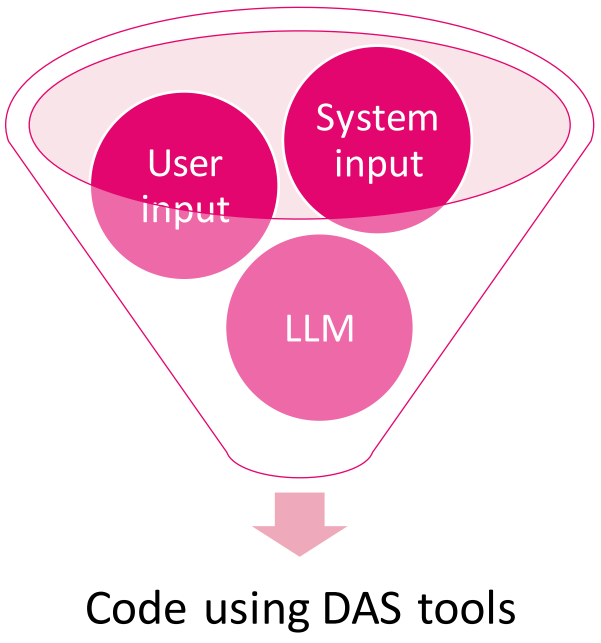


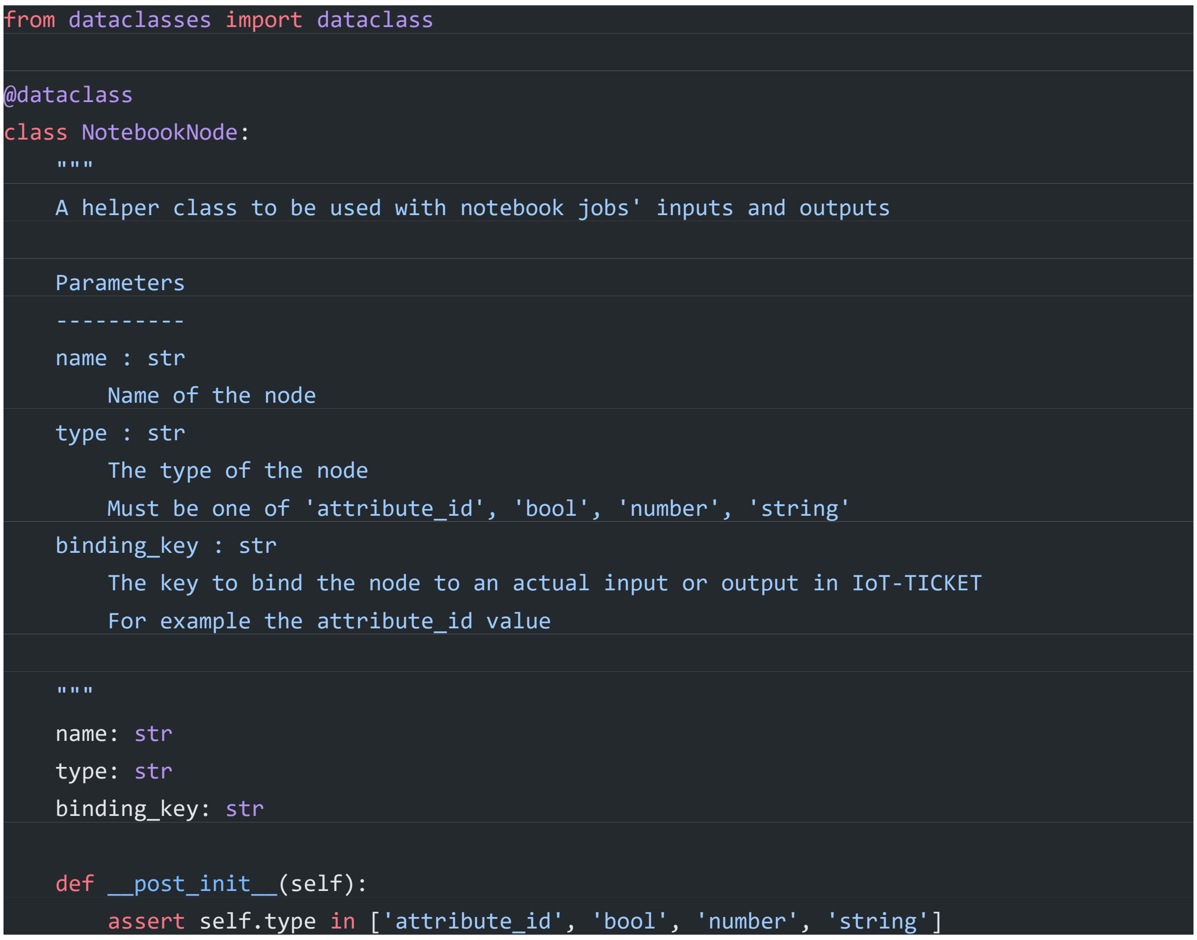
Figure 13. High-level picture of utilizing LLM to DAS.

Here system input is something we want to predefine. It is about setting fixed context to large language model. It is not required or possible for every LLMs, but we want to use that, if possible, to get better results. User input is a normal prompt-input for LLM. It can be words, sentences, sequence of sentences, code block or something like that.

In this example, system input could be following:

> NotebookNode is basically a Python dataclass with three attributes: name, type and binding_key. All are string-type and type-attribute must be one of attribute_id, bool, number or string. “inputs” and “outputs” variable are both of list-type and every item in list is an instance of NotebookNode. Using this context, define variables “inputs” and “outputs” based on the input given by the user.

Or similarly, a combination of Python-code (Figure 14) and system-prompt (continues after figure).



```python
from dataclasses import dataclass

@dataclass
class NotebookNode:
    """
    A helper class to be used with notebook jobs' inputs and outputs

    Parameters
    ----------
    name : str
        Name of the node
    type : str
        The type of the node
        Must be one of 'attribute_id', 'bool', 'number', 'string'
    binding_key : str
        The key to bind the node to an actual input or output in IoT-TICKET
        For example the attribute_id value

    """
    name: str
    type: str
    binding_key: str

    def __post_init__(self):
        assert self.type in ['attribute_id', 'bool', 'number', 'string']
```

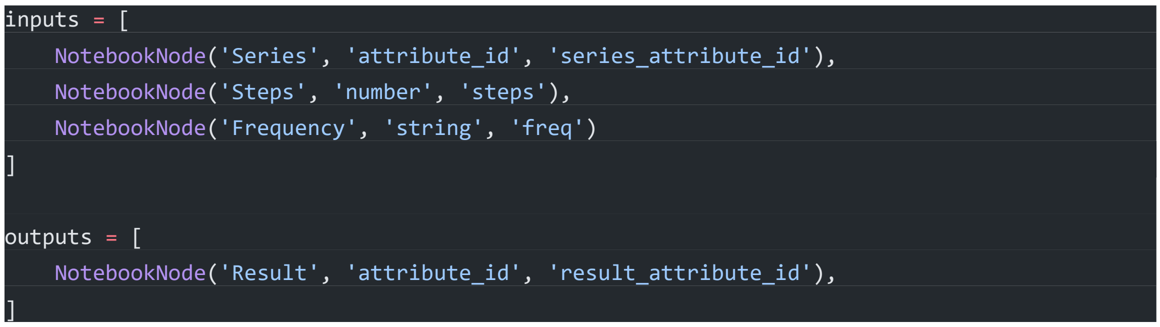
Figure 14. Example Python code as system input.

Using NotebookNode-class above, define "inputs" and "outputs" variables for the user based on user prompt. Each variable should be of type list and every item should be an instance of NotebookNode.

The user input might be the following:

I want that there are three inputs: the first should be the reference to some asset-attribute-id where the data is coming from, the second one should be how many steps forward the model should calculate and the third one is the frequency. There should only be one output which is the result of the calculation.

In the best-case scenario the output of the LLM model would be as in Figure 15.



```
inputs = [
    NotebookNode('Series', 'attribute_id', 'series_attribute_id'),
    NotebookNode('Steps', 'number', 'steps'),
    NotebookNode('Frequency', 'string', 'freq')
]

outputs = [
    NotebookNode('Result', 'attribute_id', 'result_attribute_id'),
]
```

Figure 15. Example LLM output.

3.1.2 Data Related to DAS Feature Two

This case is quite similar to what we presented in the previous section. Instead of triggering notebook jobs at Databricks, IoT-TICKET will access a general HTTP-service using POST request. HTTP-service is in most cases an Azure Function with http-trigger or something similar which allows usage of external calculation capabilities.

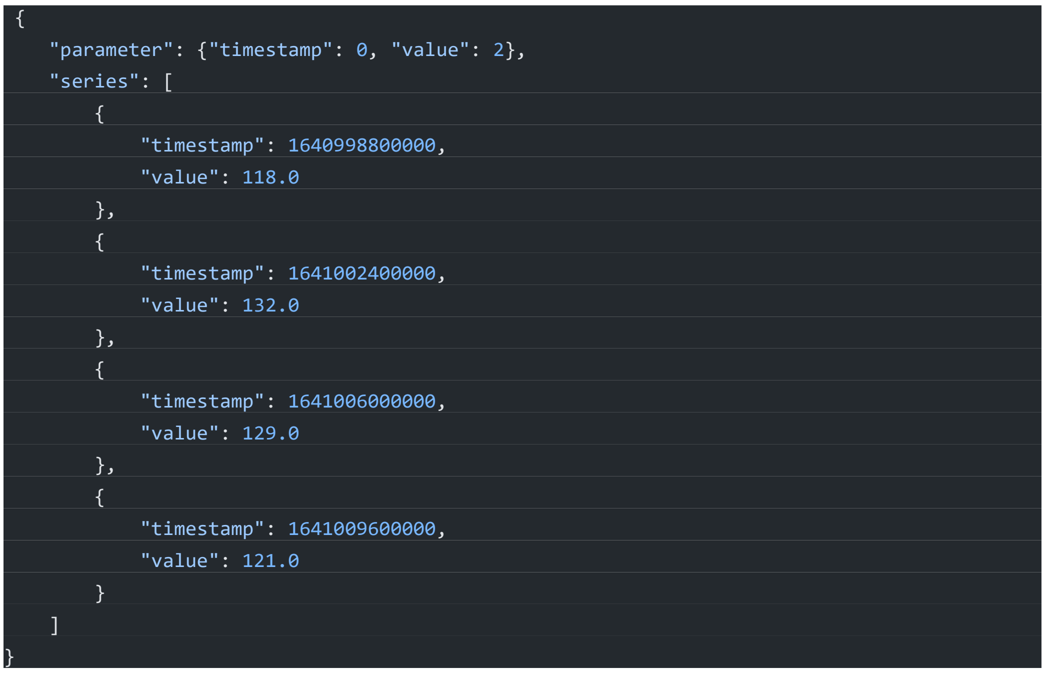
We do not repeat the whole process here since it is similar or equivalent to DAS feature one. However, this case is more complex since the (usually) JSON payload the HTTP-service takes in can be of almost any form: JSON array, JSON object, array of objects, array of arrays and so on. There is no fixed format like we had in Databricks' "notebook_params" in previous section.

The user is expected to create schema for IoT-TICKET to run the HTTP-service; like giving instructions to IoT-TICKET that “using values from this datatag and this parameter, create JSON-payload that follows given schema so that you could access the HTTP-service correctly.” The HTTP-service can be already deployed and needs only registration to IoT-TICKET, or it can be of user’s task to create such service, usually by writing a Python function that will be deployed to Azure or somewhere else. If a user creates such function, or the function code is available, it would be nice if AI would generate the schema based on the function implementation. Or furthermore, why not the function itself could be written by AI.

In the previous section, there was only one class, NotebookNode, that had effect how schema should be generated, in addition to that the “inputs” and “outputs” must be lists. Now in DAS feature 2 case, there are plenty of those. Those classes are expected to be used in different ways to mimic what is the structure of JSON array/object accepted by the service.

It is better to continue with an example to keep things simple. Let’s assume that we have a function that has two input parameters, a time series and a constant, and it returns the series multiplied by the constant. The series has two columns, timestamp and value, but only the value will be multiplied.

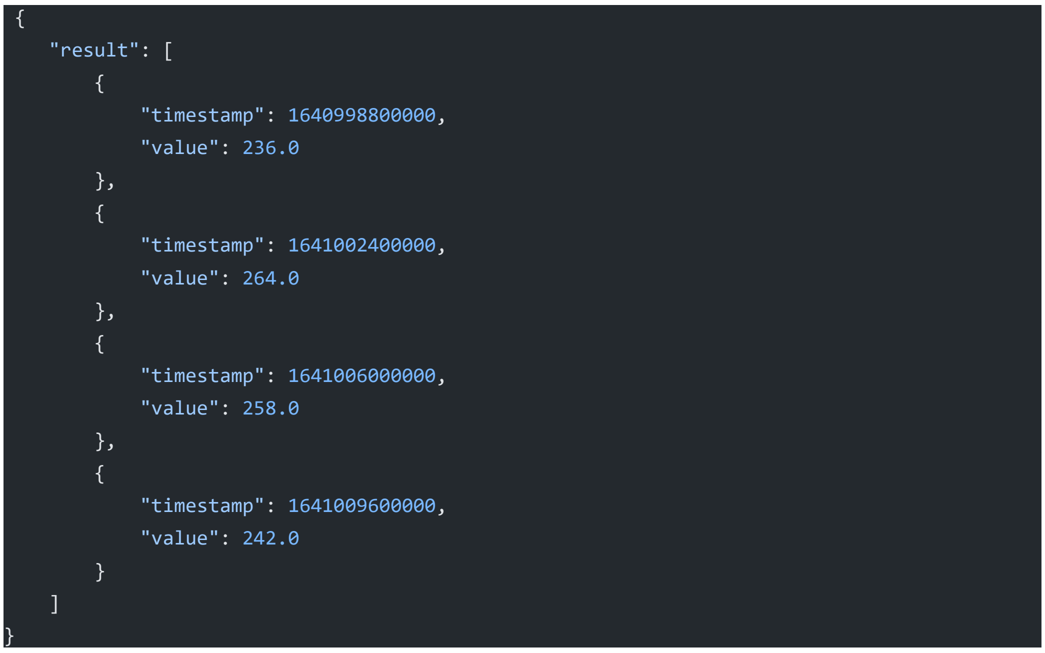
The input JSON that the function accepts is presented in Figure 16.



```
{
    "parameter": {"timestamp": 0, "value": 2},
    "series": [
        {
            "timestamp": 1640998800000,
            "value": 118.0
        },
        {
            "timestamp": 1641002400000,
            "value": 132.0
        },
        {
            "timestamp": 1641006000000,
            "value": 129.0
        },
        {
            "timestamp": 1641009600000,
            "value": 121.0
        }
    ]
}
```

Figure 16. Expected JSON payload for example function.

Here we want to point out that values that IoT-TICKET send have timestamps in every measurement or no measurement has timestamp. It is a decision made by Wapice developers. The result or response payload is shown in Figure 17.



```
{
    "result": [
        {
            "timestamp": 1640998800000,
            "value": 236.0
        },
        {
            "timestamp": 1641002400000,
            "value": 264.0
        },
        {
            "timestamp": 1641006000000,
            "value": 258.0
        },
        {
            "timestamp": 1641009600000,
            "value": 242.0
        }
    ]
}
```

Figure 17. Response payload after function calculation.

When registering such function to IoT-TICKET, the input schema and output schema should be defined so that IoT-TICKET knows what is the structure of JSON payload that should be sent and correspondingly, what is the expected structure of response payload as JSON so that the values can be parsed correctly. Using tools found in DAS-SDK, the schema is defined in Figure 18.

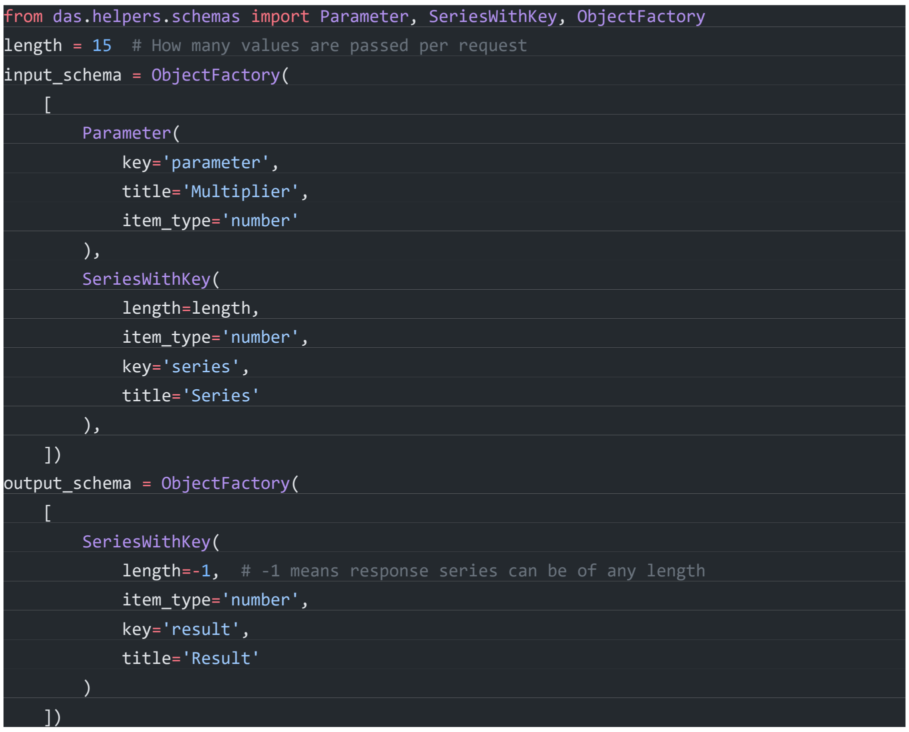

```python
from das.helpers.schemas import Parameter, SeriesWithKey, ObjectFactory
length = 15  # How many values are passed per request
input_schema = ObjectFactory(
    [
        Parameter(
            key='parameter',
            title='Multiplier',
            item_type='number'
        ),
        SeriesWithKey(
            length=length,
            item_type='number',
            key='series',
            title='Series'
        ),
    ])
output_schema = ObjectFactory(
    [
        SeriesWithKey(
            length=-1,  # -1 means response series can be of any length
            item_type='number',
            key='result',
            title='Result'
        )
    ])
```

Figure 18. Schemas before registering function to IoT-TICKET.

As discussed earlier, AI could be used both for generating function implementation and generating the input and output schemas. The input for AI could be a code sample or user-prompt using natural language. To avoid looping and closer look for every different use-case separately, we scope the target as follows: We want to use large language models to **a) generate “input_schema” and “output_schema” to be used in service registration** and **b) generate Python-function implementation that has correct amount inputs, correct structure of input-JSON and output-JSON**. In addition, only want to use classes “ObjectFactory”, “Parameter” and “SeriesWithKey” from DAS-SDK because those cover quite many use-cases, and we want to use timestamps with our values.

Using the scoping above, from now on, we talk about datasets related to DAS feature 2a and 2b. Both could be handled at the same time, but this separation was made for the sake of simplicity.

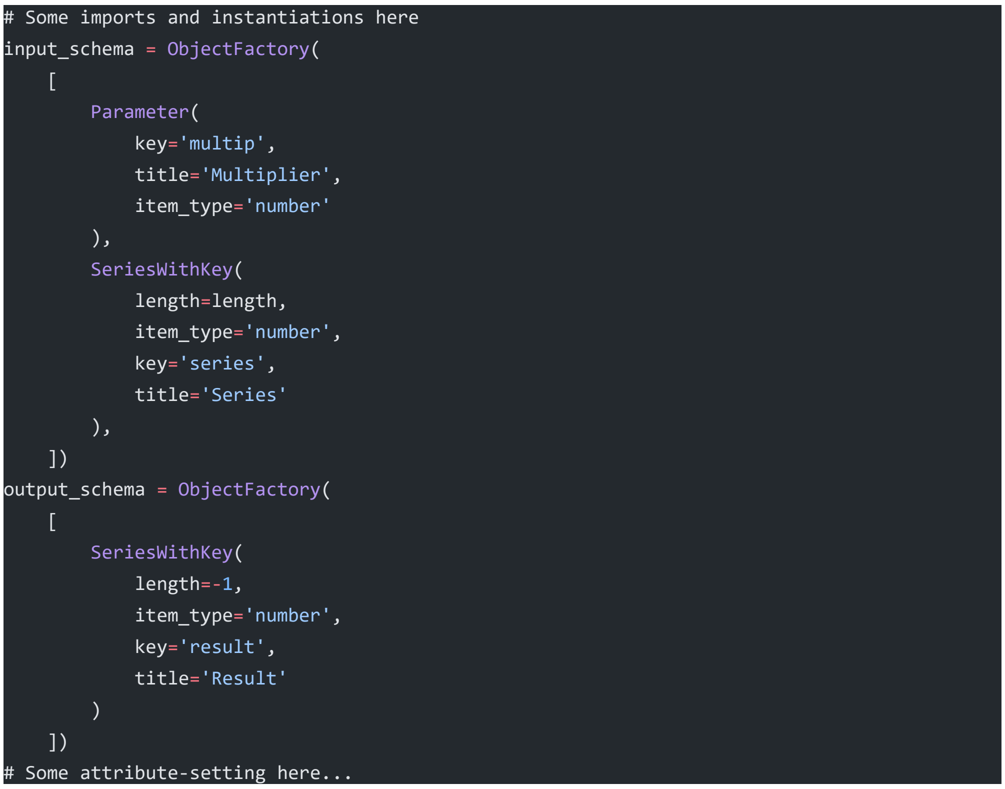
In feature **2a**, the **system-prompt** could be something like:

> *User needs to create and run registration code to register an Azure Function to IoT-TICKET. Helper tools for that could be imported from das-package made in Python. Here is an example of how it can be done: <SOME-EXAMPLE-CODE-HERE>. I want you to pay attention in creating "input_schema" and "output_schema" variables: those should be created based on the user-input. This is usually the most difficult part for the user.*

And the **user-prompt** could be:

> *I have a function that takes in one time series and a float parameter. The output is a time series.*

Here is an example of the generated output the LLM could result (Figure 19):



```
# Some imports and instantiations here
input_schema = ObjectFactory(
    [
        Parameter(
            key='multip',
            title='Multiplier',
            item_type='number'
        ),
        SeriesWithKey(
            length=length,
            item_type='number',
            key='series',
            title='Series'
        ),
    ])
output_schema = ObjectFactory(
    [
        SeriesWithKey(
            length=-1,
            item_type='number',
            key='result',
            title='Result'
        )
    ])
# Some attribute-setting here...
```

Figure 19. Example of supposed generated output from LLM using data for feature 2a.

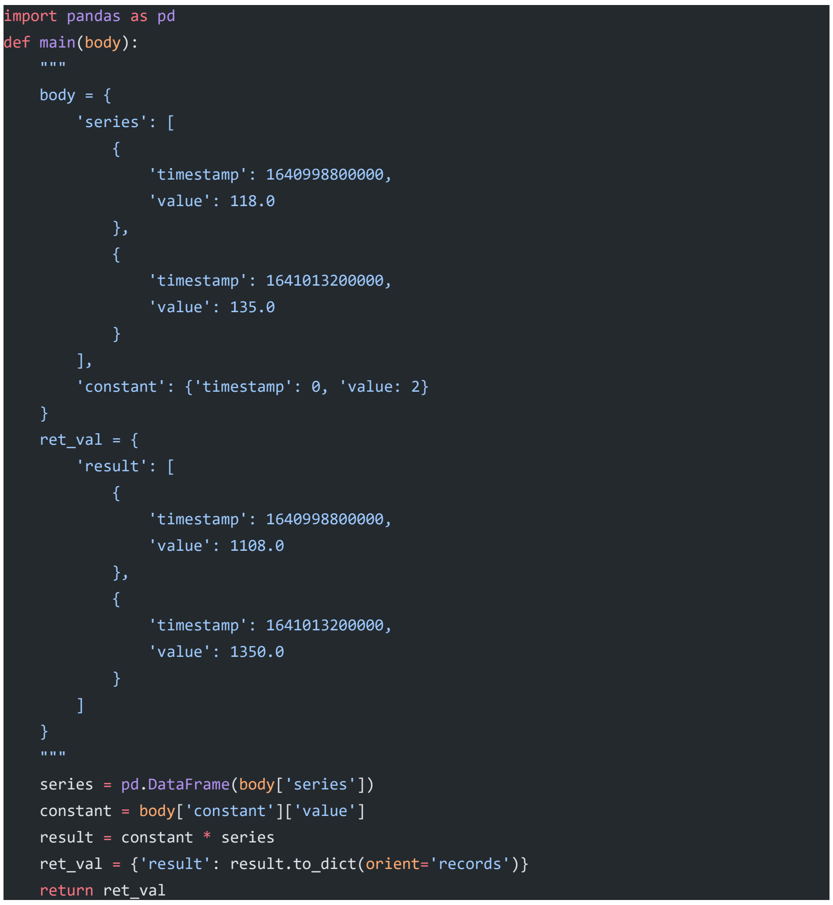
In feature **2b**, the **system-prompt** could be the following one:

> *I need you to create to user a Python function with correct imports. The function should have only one argument called "body" and it is dictionary or list. The content of "body" should be decided based on the user-input. Also, the return value format is important. Here is an example of input argument and return value: <SOME-EXAMPLE-CODE-HERE>. With these restrictions, implement the function code based on what the user wants. Add also proper docstring.*

Then the **user-prompt** could be:

> *I want a function that takes in time series and a constant. The return value is the constant * time series value.*

The expected output that the LLM could provide is presented in Figure 20.



```python
import pandas as pd
def main(body):
    """
    body = {
        'series': [
            {
                'timestamp': 1640998800000,
                'value': 118.0
            },
            {
                'timestamp': 1641013200000,
                'value': 135.0
            }
        ],
        'constant': {'timestamp': 0, 'value: 2}
    }
    ret_val = {
        'result': [
            {
                'timestamp': 1640998800000,
                'value': 1108.0
            },
            {
                'timestamp': 1641013200000,
                'value': 1350.0
            }
        ]
    }
    """
    series = pd.DataFrame(body['series'])
    constant = body['constant']['value']
    result = constant * series
    ret_val = {'result': result.to_dict(orient='records')}
    return ret_val
```

Figure 20. Generated Python function implementation related to dataset of feature 2b.

3.2 Dataset Structure and Content

3.2.1 Starting Point

If we look back in Section 2 where we presented the AI model lifecycle phases, we have so far completed the first two phases, in a way. We have gained business understanding by understanding what business problem we are trying to solve, that is, some steps in DAS-SDK usage should be easier to use. The second phase was the data understanding which we had dealt with in the previous section quite intensively.

At this point we want to point out that there existed no dataset that we could use right off. This was because the DAS-SDK tool was new and under low usage. On the other hand, since the large language models have been trained also using code repositories (e.g., from GitHub) as training data, we could be confident that those models should manage quite well even though they have not seen our private code.

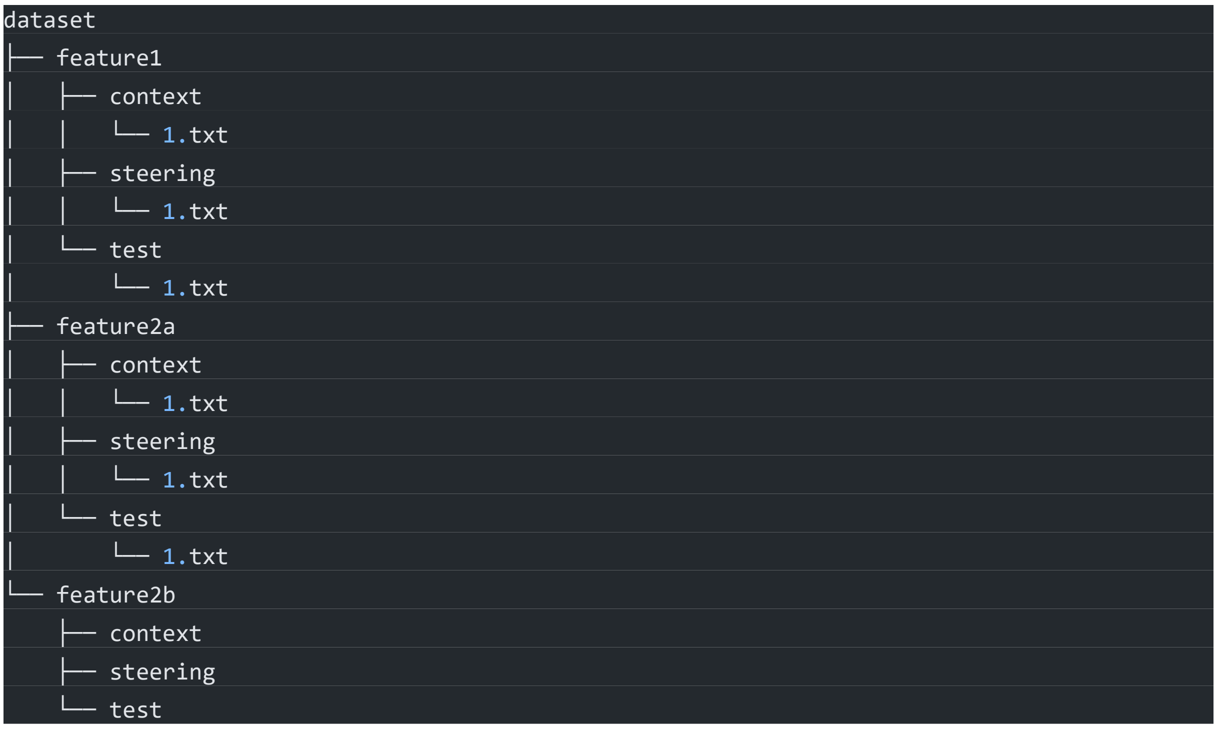
The dataset generation was done manually, using the author's competence on the subject. While writing the content of the dataset, it was easy to take care that phase three in CRISP-DM process, the data preparation phase, was done properly. Also, data division into train and test parts, part of phase four in CRISP-DM with the caveat that will be presented soon, was taken care of at the same time.

The dataset was created in such a way that it is suitable both for prompt-engineering and fine-tuning. Both of those benefit from the common part of the dataset, but some parts are separate.

At this point we want to emphasize that **the dataset is not a traditional dataset that is split into train and test parts. Instead, it is created for prompt-engineering and fine-tuning. Therefore, the naming conventions might confuse a little bit.** To relax this, we dive deeper into the dataset.

3.2.2 Folder Structure

The dataset consists of files in a folder. The structure of the folder is presented in Figure 21.



```
dataset
├── feature1
│   ├── context
│   │   └── 1.txt
│   ├── steering
│   │   └── 1.txt
│   └── test
│       └── 1.txt
├── feature2a
│   ├── context
│   │   └── 1.txt
│   ├── steering
│   │   └── 1.txt
│   └── test
│       └── 1.txt
└── feature2b
    ├── context
    ├── steering
    └── test
```

Figure 21. Dataset folder structure.

The dataset has been divided into three subfolders which all have the same structure. The **context** subfolder contains files that each have a different text content that will be used a system-prompt or task-description prompt before user-prompt.

The **steering** subfolder files contains short text that binds the context to the user-prompt. It is like “Based on the given context, I want you (AI) to generate input_schema and output_schema based on user-prompt without any natural language output”. The idea in steering is that the context can be separated from the real assignment instructions.

The **test** subfolder contains files that each has a test user-prompt and the expected label, which is a code-snippet. The test user-prompt is a natural language sentences describing the expected output of the model, and the label is corresponding code that surely works, a ground truth.

When passing data to LLM, the input is a sequence or combination of file contents from each folder in the order presented above, for example [“<content-of-context/1.txt>”, “<content-of-steering/2.txt>”, “<content-of-test/3.txt>”]. By making all possible combinations of the possible sequences, we have a dataset for a feature. By mixing, we can see how different contexts, steerings and user-prompts affect the generated output.

All in all, the size of the dataset is presented in Table 1.

Table 1. Size of the dataset in number of files.

FEATURE	CONTEXT	STEERING	TEST
1	5	5	10
2A	5	5	10
2B	5	5	20

The content of the dataset has been formed manually. The author spent many hours trying to find a starting point that is good enough to generate reasonable outputs. This starting point is identified by the file name “1.txt” for each case. After that, the author altered the content to get file “2.txt” and so on.

The dataset adapts the way the author is thinking and writing, of course. To get more unbiased dataset, more data from different test-users should be collected. Since this is not a scientific publication and since it is not possible to gather such data in the scope of this bachelor thesis, we are satisfied with the current dataset.

3.2.3 Data Samples

The first part is related to DAS feature 1; it is about generating inputs and outputs to Data-bricksJob-class without extra stuff like imports and instantiation etc. A sample of a **context file** is presented in Figure 22.

```
from dataclasses import dataclass

@dataclass
class NotebookNode:
    \"\"\"
    A helper class to be used with notebook jobs' inputs and outputs

    Parameters
    ----------
    name : str
        Name of the node
    type : str
        The type of the node
        Must be one of 'attribute_id', 'bool', 'number', 'string'
    binding_key : str
        The key to bind the node to an actual input or output in IoT-TICKET
        For example the attribute_id value

    \"\"\"
    name: str
    type: str
    binding_key: str

    def __post_init__(self):
        assert self.type in ['attribute_id', 'bool', 'number', 'string']
```

Figure 22. Sample of DAS feature 1 context file.

A sample from **steering** folder is following: “User wants to define “inputs” and outputs” variables. Both variables should be a list of NotebookNode-instances defined before.”

A sample from the **test** folder is presented in Figure 23. The user-prompt and label are separated using “---”.

```
I want one data input and one data output.
---
inputs =  [
    NotebookNode('Series', 'attribute_id', 'series_id'),
]
outputs =  [
    NotebookNode('Result Series', 'attribute_id', 'result_series_id')
]
```

Figure 23. Sample of test file of DAS feature 1.

To save space here, we present only **test** data for feature 2a and 2b. The corresponding contexts and steerings are present in the original dataset.

The feature2a folder is related to DAS feature 2 and the idea is to generate a registration code for Azure Functions. The sample file of the test folder is presented in Figure 24. The first part before separator "---" is the user-prompt and the latter part is the label, i.e., the expected answer.

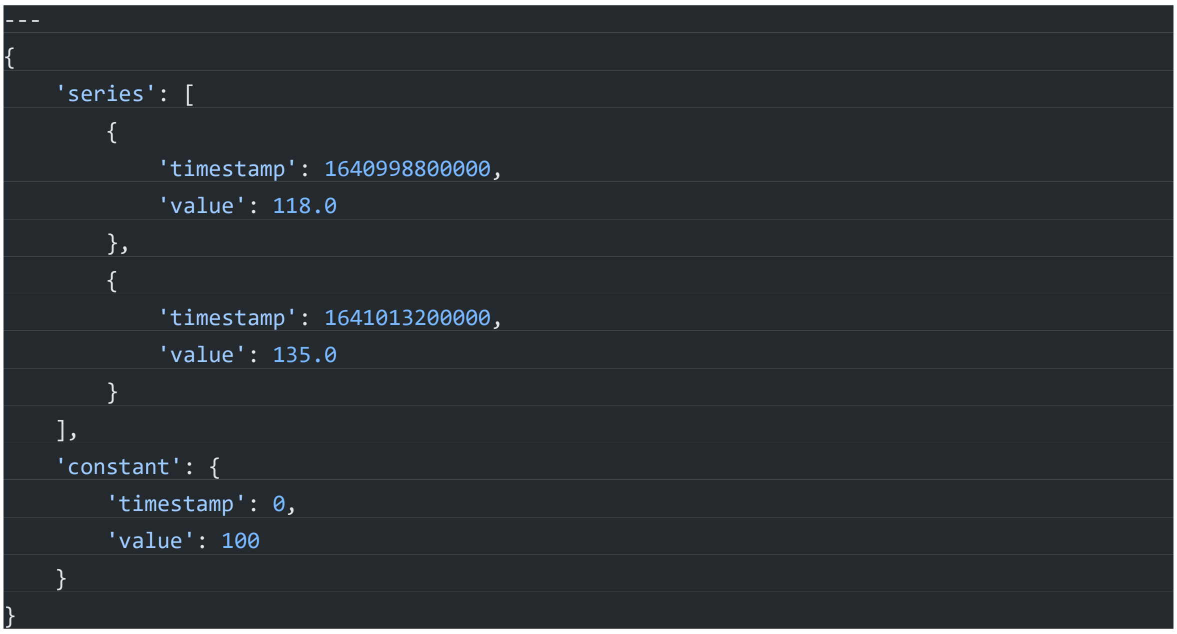
```
I have a function that has one series-type input of length 4, and there are four outputs:
maximum, minimum, length and average. I need a code to register it to IoT-TICKET.
---
from das import IoTClient
from das.helpers import AzureFunction
from das.helpers.schemas import Parameter, ObjectFactory, SeriesWithKey

function = AzureFunction(...)
function.input_schema = ObjectFactory(
    [
        SeriesWithKey(
            length=4,
            item_type='number',
            key='data',
            title='Series')
    ])
function.output_schema = ObjectFactory(
    [
        Parameter(key='max', title='Maximum', item_type='number'),
        Parameter(key='min', title='Minimum', item_type='number'),
        Parameter(key='len', title='Length', item_type='number'),
        Parameter(key='avg', title='Average', item_type='number')
    ])
func = function.as_dict()
func['request_contains_timestamps'] = True
func['response_contains_timestamps'] = True
iot = IoTClient()
iot.register_service(func)
```

Figure 24. Sample file from a feature2a/test folder.

The feature2b-folder is also related to DAS feature 2, but now we want to generate a whole function. An example file of test folder is presented in Figures 25 and 26. There are three parts: user-prompt, label and test payload. The latter one will be used to check if the generated function will handle the test payload.

```
I want the implementation of a function, that has two inputs: series-type and constant pa-
rameter. The function returns input-series multiplied by input-constant as a result.
---
import pandas as pd
def main(body):
    """
    body = {
        'series': [
            {
                'timestamp': 1640998800000,
                'value': 118.0
            },
            {
                'timestamp': 1641013200000,
                'value': 135.0
            }
        ],
        'constant': {
            'timestamp': 0,
            'value': 100
        }
    }
    ret_val = {
        'result': [
            {
                'timestamp': 1640998800000,
                'value': 1108.0
            },
            {
                'timestamp': 1641013200000,
                'value': 1350.0
            }
        ]
    }
    """
    series = pd.DataFrame(body['series'])
    constant = body['constant']['value']
    result = series * constant
    ret_val = {'result': result.to_dict(orient='records')}
    return ret_val
```

Figure 25. User-prompt and label from feature2b/test folder file.



```
---
{
    'series': [
        {
            'timestamp': 1640998800000,
            'value': 118.0
        },
        {
            'timestamp': 1641013200000,
            'value': 135.0
        }
    ],
    'constant': {
        'timestamp': 0,
        'value': 100
    }
}
```

Figure 26. Test payload from feature2b/test folder file.

3.2.4 Division to Prompt-Engineering and Fine-Tuning

As discussed before, some parts of the dataset can be used in prompt-engineering and some parts in fine-tuning. Next, we will find out what that division is.

The context and steering folder are being used in prompt-engineering; either as system-prompts in ChatCompletion endpoints or as merged in a pre-defined user-prompt in Completion endpoints. The user-prompts and labels from test folder files are also used in prompt-engineering, but the test payloads are not.

In fine-tuning, we do not use context and steering folders. We only use data from the test-folder. Here the reader might get confused, due to the naming convention of such folder. But anyway, the user-prompts and labels are being used in fine-tuning, which we will explain next.

Depending on the model and how the fine-tuning API is used, the data training data format may differ. OpenAI's models that can be fine-tuned require that the training data is of JSONL-format (OpenAI, *Prepare training data*). The format is presented in Figure 27.

```
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
...
```

Figure 27. JSONL formatted training data example.

So, using the format above, *prompt* is the user-prompt and *completion* is the label, both found in the files in the test-folder.

Since our dataset was created manually and it is hard to create it big enough, we try fine-tuning only on feature 2b whose test-folder content is the biggest one. It is still a small dataset but at least something. This results in 20 samples of data and doing 80-20 division we result 16 samples of training (fine-tuning) data and 4 samples of validation data.

3.3 Available Model Catalog

There are different actors that are training and publishing large AI models, including large language models. Different actors focus on different purposes and use-cases. Models can be open or closed, created by open-sourced communities versus companies.

In this subsection we present the most famous actors and their models. We have chosen only such actors whose models are focused on generating text. Therefore, some interesting actors and their models, e.g., BERT from Google, are left out. For a comprehensive list of language models, we ask the reader to see list of large language models in Wikipedia (Wikipedia article, n.d.).

Most of the models are not open. Training phases of the models have been computationally very heavy and thus expensive. Also, the inference phase, where the model is being used by running data against it, is quite expensive. To mitigate the cost, many actors ask for money from it. Only some actors like EleutherAI who is considered as an open-source version of OpenAI, have big enough community and resources to develop models. Due to its resource intensiveness nature and expensiveness, only big actors have capabilities to compete on this area. This means that those actors have huge power and thus responsibility to keep things ethical.

3.3.1 OpenAI

OpenAI company is probably the most famous company that develops AI models. They have different sets of models and families of models for different tasks. A model's lifetime cycle is short, and it once happened that the author got familiar with a certain model just to find out it deprecated a couple weeks later.

GPT-4

GPT-4 model family is a set of models that improve on GPT-3.5 and can understand as well as generate natural language or code (OpenAI documentation page for models, n.d.). It is the newest GPT model that is under limited beta stage, available only to selected users. Tasks where the model is good at or designed to, are text and chat completion, text edition, content generation and summarization (ibid). The access to this model is available on ChatGPT Plus web app-subscribers, selected free tier users and selected partners using Azure OpenAI service.

GPT-3.5

The GPT-3.5 model family is a couple months older version of GPT-4 and meant for similar tasks as its successor. The most used models from this family are GPT-3.5-turbo and GTP-3.5-turbo-0301 in addition to couple fine-tuned davinci-based models (explained on GPT-3 subsection). It is easy to get access to these models. For example, ChatGPT free tier web-app uses GPT-3.5-turbo and other models can be used through OpenAI Playground for developers or through API.

GPT-3

The GPT-3 model family consists of fine-tunable base models called ada, babbage, curie and davinci plus some fine-tuned models based on the base models. Family was released in 2020 and it is considered a little old fashioned. On the other hand, there are no newer base models that could be fine-tuned (OpenAI documentation page for models, n.d.), so this family is still relatively relevant. Models from this family can be accessed through API or OpenAI Playground.

Codex

Models in Codex family can “understand and generate code, including translating natural language to code” (OpenAI documentation page for models, n.d.). The most familiar model is code-davinci-002 and it is based on davinci-model from GPT-3 family.

However, codex models are now deprecated and removed from OpenAI’s public access. Fortunately, one can still find and use this model from Azure OpenAI service, which is available for the author.

Other

The rest of the models from OpenAI catalog are not interested in the scope of this thesis. In general, these models, however, have great importance and different use-cases.

For image generation and image editing tasks, there is a model named **Dall-E**, which understands also natural language prompts. For converting audio into text for tasks like transcription and translation, one should get familiar with **Whisper**. To convert text into a numerical form for tasks like search, clustering, recommendations, anomaly detections and classification, there are **Embeddings** models, and for content moderation tasks there are **Moderation** models. (OpenAI documentation page for models, n.d.).

3.3.2 Bing AI

Bing is a search engine by Microsoft that was launched in 2009. Recently, in February 2023, Microsoft announced the new AI-enable Bing, which is using OpenAI’s GPT-models presented above. (Mehdi, 2023).

After Microsoft’s announcement, GPT-4 from OpenAI was published and soon after that it was revealed that the Bing is using GPT-4 as a backend (Lardinois, 2023).

On top of the GPT-4 model, Microsoft has added a model called Prometheus, but it is more a collection of capabilities and technologies to leverage GPT-4 more powerful than a pure model. Furthermore, Bing search ranking engine is now powered by the AI model thus giving more accurate and relevant search results than before. Bing also provides a chat experience which combines searching, browsing and chatting into one experience. (Mehdi, 2023).

At the time of writing, March 2023, the Bing AI can be used only by joining to waitlist. Later Bing AI will be part of Bing search but also an integrated part of Microsoft Windows search experience. (idis).

Bing AI currently has no API for developers so possible usage of this model into our research topic must be done through web browser. Therefore, it cannot be utilized as part of the DAS-SDK currently, but things might change in future, so we don’t reject this model right away.

3.3.3 Meta AI

Meta, also known as Meta Platform Inc., and formerly as Facebook Inc., has also released their large language model recently, on February 24, 2023. Their model is called LLaMA, name coming from Large Language Model Meta AI. (Model release page at Meta website).

LLaMA is a state-of-the-art foundation model that is designed for AI researchers. It is smaller and more performant than e.g., GPT-models from OpenAI, and thus accessible by the researchers that do not have large amounts of infrastructure to study these models, making the access more democratizing. (idis).

Meta AI provides access to four (4) foundation models of sizes 7B, 13B, 33B and 65B parameters. As the naming convention “foundation” says, these models are not meant to be ready to be used in applications but instead, they need to be fine-tuned by retraining them on application-specified dataset. (idis).

This model is very interesting in different applications due to its smaller size and ability to be fine-tuned. Unfortunately, in the scope of this thesis

3.3.4 Github Copilot

Github Copilot is a helper tool for programmers. It can be used directly from selected editors including Visual Studio, Visual Studio Code, Neovim or JetBrains IDEs. The model behind is OpenAI's Codex that is accessed over the internet. (Github Copilot web page).

That said, Github Copilot cannot be considered as a separate large language model. It is, however, an interesting concept due to how the Codex model is accessed. Using the Copilot in IDE, the context and changes in code file are automatically noticed by Copilot and send to Codex model in the backend process of IDE. To guide the Copilot and the model behind, the user can write hints and orders in comment-lines and the Copilot fills the rest. Copilot can also fill the gaps left by developer. (idis).

What is interesting about Copilot is that the usage of Codex model is optimized in the sense of prompting. When accessing Codex through API or Playground, the user must give the context and input prompts to instruct the model. It is not clear that the user could format inputs such that the model's response is good enough or the best. There are best practices and examples of how to get the best out of the model, but still, it is not an easy task. Thus, letting Copilot do the job for prompting, and possibly get the best out, is a wise choice.

3.3.5 Other

There are of course other models than those presented above. Some are listed on Wikipedia (Wikipedia, Large language model) but not all. Some models that have become irrelevant since better ones have been developed, as is the case e.g., for OPT model family by Meta AI. In their research paper, Touvron et al. (2023) showed that the new LLaMA model family outperforms OPT-family models by Meta AI in results but also in energy usage.

The development of large language models is very fast and models that were on top notch one day can be outdated very quickly. Therefore, it is not reasonable to stick with some models or model families, more important is itself updated what is going on.

While writing this sentence, March 20, 2023, we are waiting for Google's answer to the development of large language models. Google has some potential with its models PaLM and LaMDA. Also, NVIDIA together with Microsoft have their joint model called Megatron-Turing (Smith et al.).

In the next section we focus on selecting proper models for further testing. The scope of this thesis is to make some comparisons between the models' performance, but, above all, to investigate how the large language models overcome certain code-generation related tasks. Therefore, we have not tried to find every possible model available or felt disappointed if certain interesting models were not selected for further investigation.

3.4 Model Selection for Test Runs

The following models were selected on test runs. For **prompt engineering**: GPT-3.5-turbo, code-davinci-002, GPT-4 and Bing AI. For **fine-tuning**: curie.

Another interesting models would have been LLaMA by Meta AI and GitHub Copilot. The author got access to LLaMA models on 18 April 2023, after spending couple of weeks on the waiting list. The lightest 7B model weights were downloaded resulting in 13 GB of files. Soon it was found out that the GPU memory requirement for the model was about 28 GB, so the model usage was out of the question with the local resources and lack of ready-to-go development environment like Azure OpenAI service. Furthermore, Github Copilot was left out due to its pricing and the need of a credit card.

The instances of the models, except Bing AI, were deployed to Azure OpenAI service provided by the client of this thesis. This made the model usage faster since we did not have to compete the resources between other users. Bing AI was accessed using the author's personal Microsoft account, restriction made by Microsoft.

At this point we want to emphasize that our dataset is not big enough for fine-tuning. As discussed in Section 2.3.2, there should be at least thousands of labeled examples that should be used to fine-tuning. Our dataset has only a few of those and it would have been huge effort to create such a big train dataset. However, the fine-tunable curie model was included because we wanted to study how fine-tuning worked, fine-tuning was available as a service, and to create a training framework for our use-case. The was also a better performing and fine-tunable model davinci, but it was not available in Azure OpenAI service at the moment of writing.

In fine-tuning, we only focused on DAS feature 2 and its function generation abilities instead of creation of registration code. That is, we used dataset related to feature 2b.

3.5 Scoring

When we get the outputs of the models, also known as **completions**, we need a tool that says how good the output is with respect to testing data (label). Since the outputs are pieces of code, we should carefully find the proper **score** function for comparison. Since this thesis is not a scientifical research, we take some liberties on this. We define our score function for each feature as follows:

- Feature 1
  - 1p (point) if output code raises no SyntaxError when executing it.
  - 1p if variables named “inputs” and “outputs” are set during execution.
  - 1p if “inputs” and “outputs” variables have same length as corresponding variables in test data.
  - 1p if items inside “inputs” and “outputs” variables have same type-attributes as corresponding variables in test data.
- Feature 2a
  - 1p if output code raises no SyntaxError when executing it.
  - 1p if a variable named “func” is set during execution.
  - 1p if “func” variable has same keys as corresponding variable in test data.
  - 1p if “func” variable has also same deep structure as corresponding variable in test data.
- Feature 2b
  - 1p if output code raises no SyntaxError when executing it.
  - 1p if function implementation raises no SyntaxError when compiling it.
  - 1p if function named “main” is defined.
  - 1p if function main has only one argument “body”
  - Extra 1p: generated function raises no exception when running it using test payload.
    - This is extra since we do not expect fully implemented functions in every test file.
    - The test payload is the same one mentioned in the label’s docstring.

In general, there are code-similarity and program-similarity tools that give reports how close two code-snippets are, but we omit those due to small size of dataset and the lack of scientific approach.

4 Test Runs and Findings

4.1 Test Bench

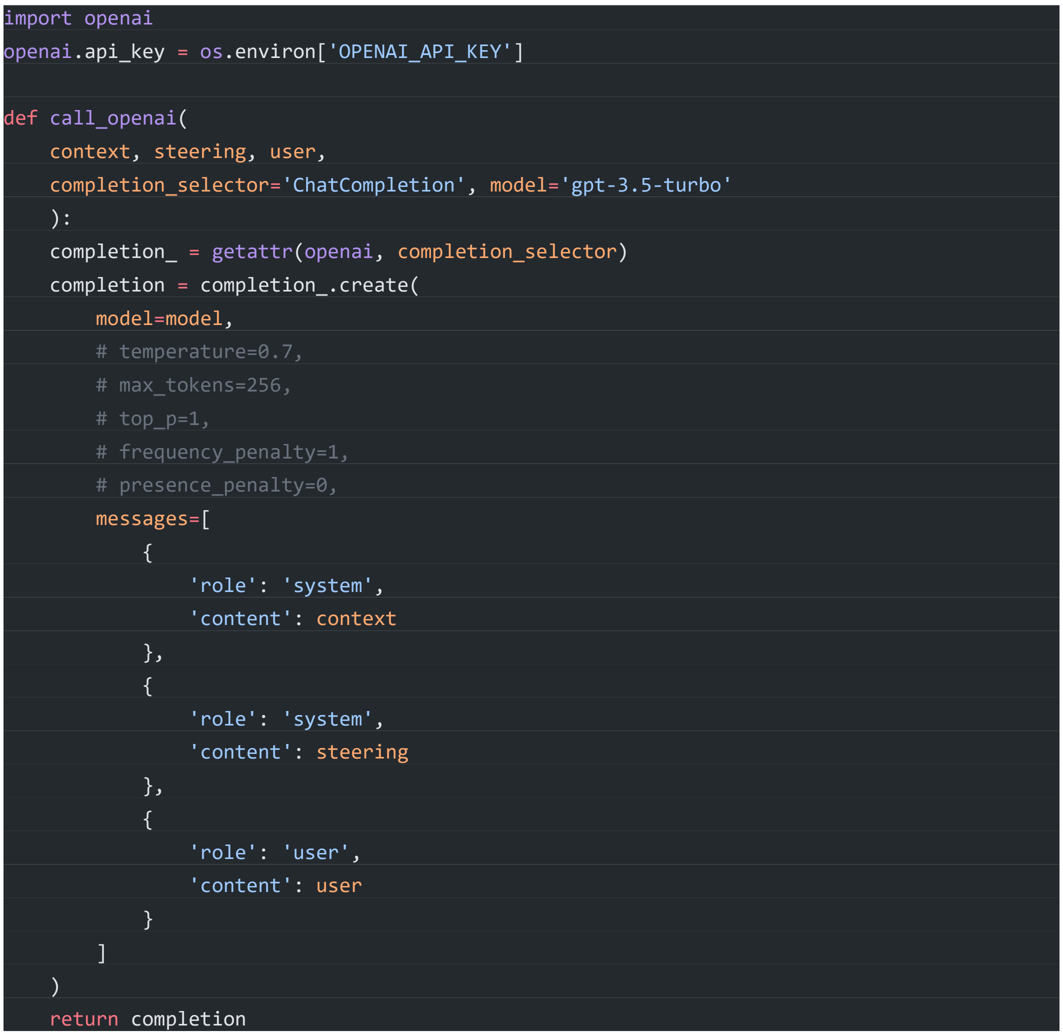
Now we are ready to run tests against our dataset. To make test runs and scoring as automated as possible, a common test bench was designed, and it was implemented in different ways depending on how the LLM could be accessed. OpenAI has a Python library to access their models, so it was nice and easy to implement the bench using that library and other Python code. The other extremity was Bing AI. It has no API, so we did many manual copy-pasting texts back and forth to get the results and scores.

The dataset was read from files and a combination grid was formed. Figure 28 shows the procedure for feature 2a.

```python
import itertools
import pathlib

context_prompts = []
for f in pathlib.Path('dataset/feature2a/context/').iterdir():
    context_prompts.append({
        'file_name': f.parent.name + '/' + f.name,
        'content': f.read_text()
    })
steering_prompts = []
for f in pathlib.Path('dataset/feature2a/steering/').iterdir():
    steering_prompts.append({
        'file_name': f.parent.name + '/' + f.name,
        'content': f.read_text()
    })
test = []
for f in pathlib.Path('dataset/feature2a/test/').iterdir():
    content = f.read_text()
    user_prompt, label = content.split('---')
    test.append({
        'file_name': f.parent.name + '/' + f.name,
        'user_prompt': user_prompt,
        'label': label
    })
all_combinations = []
for combination in itertools.product(context_prompts, steering_prompts, test):
    all_combinations.append(combination)
```

Figure 28. Dataset read in test bench.

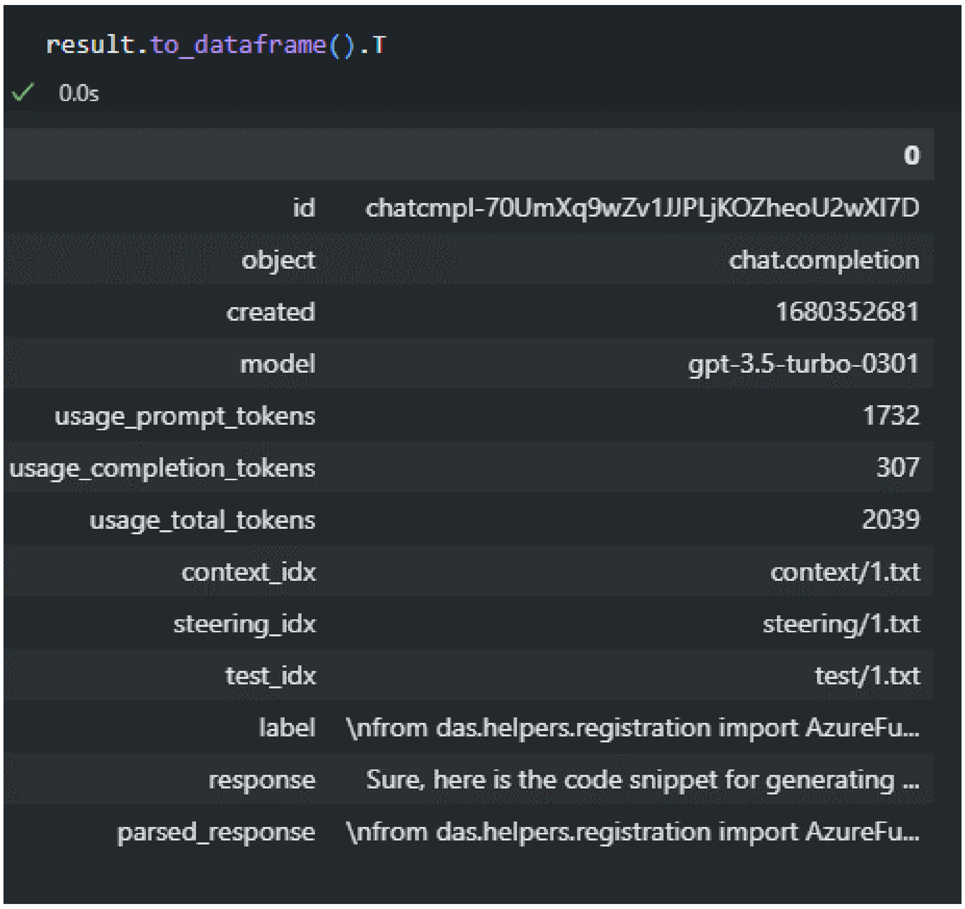
Then, using the content in “all_combinations” variable, the model was called. In OpenAI cases, the model call was the following procedure (Figure 29).



```python
import openai
openai.api_key = os.environ['OPENAI_API_KEY']

def call_openai(
    context, steering, user,
    completion_selector='ChatCompletion', model='gpt-3.5-turbo'
    ):
    completion_ = getattr(openai, completion_selector)
    completion = completion_.create(
        model=model,
        # temperature=0.7,
        # max_tokens=256,
        # top_p=1,
        # frequency_penalty=1,
        # presence_penalty=0,
        messages=[
            {
                'role': 'system',
                'content': context
            },
            {
                'role': 'system',
                'content': steering
            },
            {
                'role': 'user',
                'content': user
            }
        ]
    )
    return completion
```

Figure 29. GPT-3.5-turbo model access through public API using OpenAI python library.

The model was accessed multiple times by looping iterables in “all_combinations” variable. The loop was a Python loop or human loop, depending on how the model can be accessed. Then the results were stored in an instance of RunResult-class, that has certain helper methods to be used in scoring. For example, a parser method was implemented; it parses the python code from a response block that contains natural language text and code blocks. The following example shows what information was stored in a result (Figure 30).



```
result.to_dataframe().T
✓ 0.0s
                                                                    0
                       id   chatcmpl-70UmXq9wZv1JJPLjKOZheoU2wXI7D
                   object                          chat.completion
                  created                               1680352681
                    model                       gpt-3.5-turbo-0301
      usage_prompt_tokens                                     1732
  usage_completion_tokens                                      307
       usage_total_tokens                                     2039
              context_idx                            context/1.txt
             steering_idx                           steering/1.txt
                 test_idx                               test/1.txt
                    label   \nfrom das.helpers.registration import AzureFu...
                 response     Sure, here is the code snippet for generating ...
          parsed_response   \nfrom das.helpers.registration import AzureFu...
```

Figure 30. Example of RunResult for one run (transposed for more compact view).

Finally, the scoring function was called to get score for the result, which is “parsed_response”. The scoring function in this case (feature 2a) checks if result has SyntaxErrors, checks if the code can be run itself without exception and finally compares the result to the label by comparing the dictionaries they form. As a reminder, the (other) scoring methods were presented in Section 3.5. The scores were added at the end of the result dataframe as a new column.

4.2 Runs

4.2.1 General Observations

In general, all test runs went well and there were no serious problems. Due to occasional rate limiting errors and missing responses, which were handled by trying again, all went well. Running times were not measured since the models are constantly under heavy load and sometimes the response should be waited quite long.

After the test runs, when the first impressions from the results were investigated, a couple of problems were found. The first problem was that the generated outputs, or more precisely, the parsed code blocks from generated outputs, included the usage of Python built-in function "input". It meant that in the scoring phase, the code execution would halt, and the input should be made by the author. This was handled by detecting those code-lines by using Python's mock-library during the execution and giving zero points to result.

Another problem was related to the data in the test-folder. The author had malformatted some labels such that they did not work as expected. The labels were code-snippets and those should have been executed before the test run phase. The results that had something to do with malformatted data were removed from the result dataset. Luckily, only three labels from feature 2a and one label from feature 2b were affected, so it was not such a big concern.

All in all, total model access count for GPT-3.5-turbo, GPT-4 and code-davinci-002 together, was 6000. After removing non-usable results, the count lowered to 5400. Results from fine-tuned curie model and Bing AI were examined only from qualitative perspective.

4.2.2 GPT-3.5-turbo and GPT-4

All test runs used chat-completion endpoint, see Section 2.4 for explanation. Both models were accessed in two different test runs. The first run had a temperature value of 0.4 and the second run had value 0.05. Higher temperature means that outputs are more random while lower values make the output more focused and deterministic (OpenAI documentation page for models, n.d.). All data from the dataset, i.e., data related to every feature, was used.

4.2.3 code-davinci-002

Code-davinci-002 model provides no chat-completion endpoint, only completion endpoint. Therefore, it was not possible to use the message style presented in Figure 29 above. To mitigate this, messages were concatenated to format *"<context>. <steering>. Based on that, the user wants that <user>. So my answer that include only code is…"*. This format was chosen because it provided best results for sample data over another formats.

There were multiple separate runs with the parameter combinations shown in Table 2. Before automated runs, the combinations were predefined using some samples from datasets to avoid extra costs, repetitions in answer and in general, to see if results looked good enough. Wrong combinations might give non-usable answers, so some preparation needed to be done.

Table 2. Parameter combinations for code-davinci-002 runs.

Feature	**Run**	**Temperature**	**Frequency penalty**	**Max tokens**
1	1	0.4	0	600
	2	0.05	0.5	200
2a	1	0.4	0	600
	2	0.05	0.5	200
2b	1	0.4	0.5	700
	2	0.05	0.5	700

Here frequency penalty is a parameter that "can be used to reduce the likelihood of sampling repetitive sequences of tokens" (OpenAI, API reference page). Max tokens is the maximum number of tokens to be generated in completion or chat-completion endpoint.

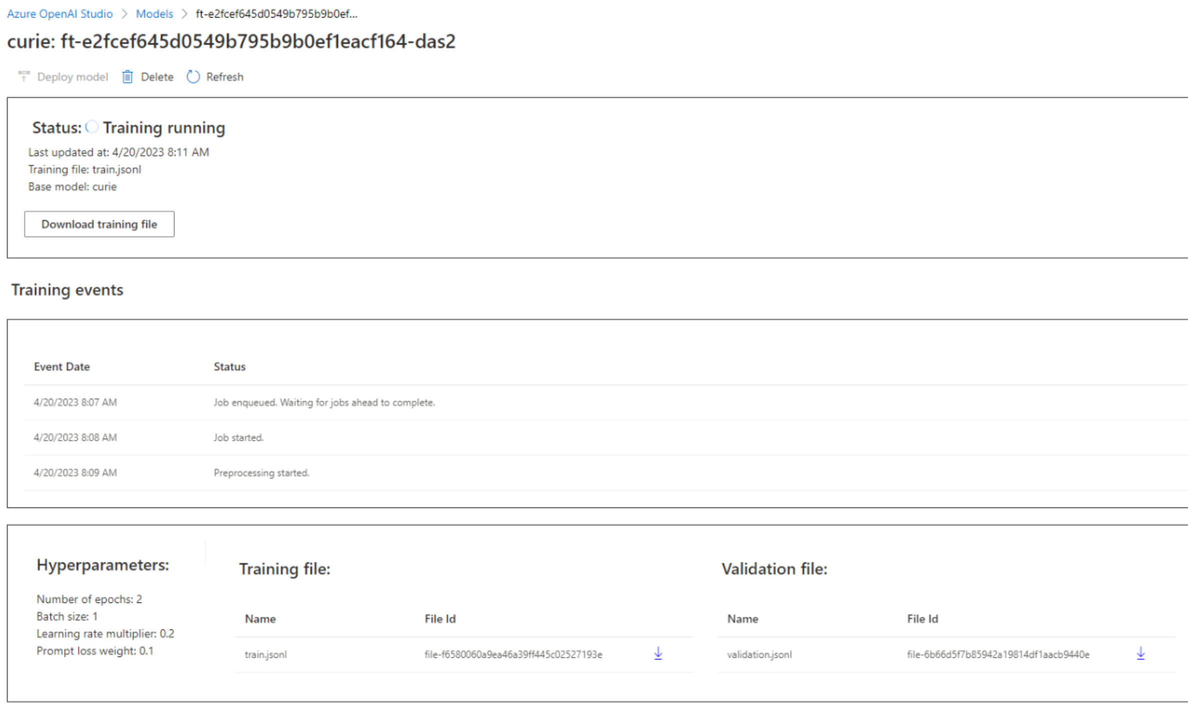
4.2.4 Curie (Fine-Tuned)

The curie model from OpenAI is the only model in this thesis that was fine-tuned. We concentrated only on dataset related to feature 2b. That was the Python function generation part of the whole feature 2. Otherwise, if we had also taken the registration code generation (feature 2a), the size of the dataset for fine-tuning would have been too large to populate from scratch, while this is still the case.

Fine-tuning

The dataset was prepared so that it could be used for fine-tuning. All the files from subfolder feature2b/test/* we changed to JSONL-format discussed in Section 3.2.3. Also, a separator and stop sequences were added to follow the guidelines from OpenAI fine-tuning instructions (OpenAI, Fine-tuning). No data from context or steering folder were used, naturally, as discussed in Section 3.2.3. The resulting dataset was divided into 80-20 divisions resulting in training data of 16 samples and validation data of 4 samples.

The fine-tuning process was started from Azure OpenAI service, see Figure 31.


Azure OpenAI Studio > Models > ft-e2fcef645d0549b795b9b0ef...
Deploy model
Delete
Refresh
Download training file

curie: ft-e2fcef645d0549b795b9b0ef1eacf164-das2

Status: Training running
Last updated at: 4/20/2023 8:11 AM
Training file: train.jsonl
Base model: curie

Training events

Event Date	Status
4/20/2023 8:07 AM	Job enqueued. Waiting for jobs ahead to complete.
4/20/2023 8:08 AM	Job started.
4/20/2023 8:09 AM	Preprocessing started.

Hyperparameters:
Number of epochs: 2
Batch size: 1
Learning rate multiplier: 0.2
Prompt loss weight: 0.1

Training file:

Name	File Id
train.jsonl	file-f6580060a9ea46a39ff445c02527193e

Validation file:

Name	File Id
validation.jsonl	file-6b66d5f7b85942a19814df1aacb9440e

Figure 31. Fine-tuning of Curie model is on running phase in Azure OpenAI service.

After some moments the fine-tuning succeeded. The fine-tuned model behaves similarly to any other OpenAI model so it can be used from playground or through the API, but with the restriction that only completion endpoint can be used.

The training and validation losses are shown in Figure 32. As one can easily see from it, it is not good, and we already know the reason for that (small dataset).

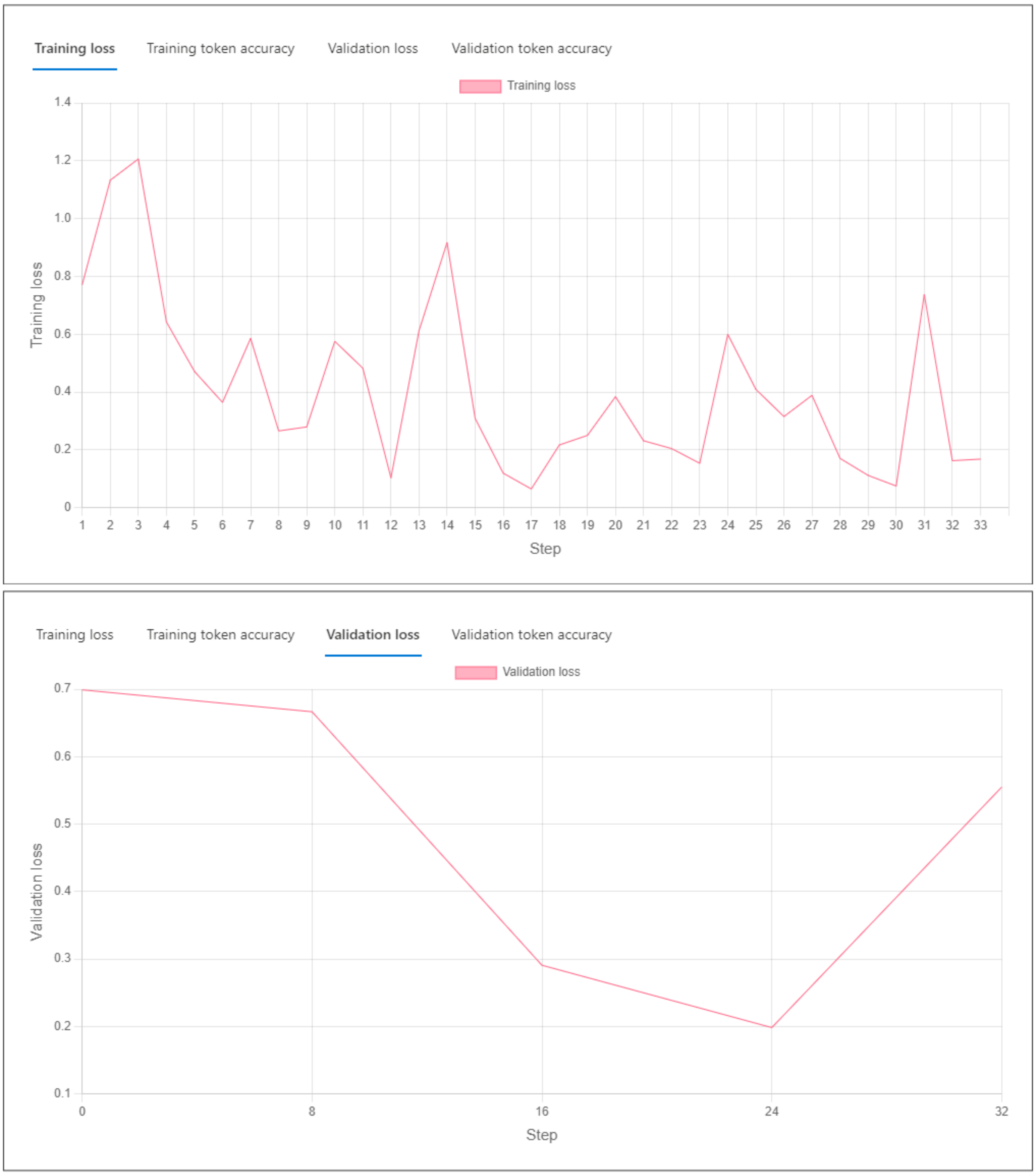
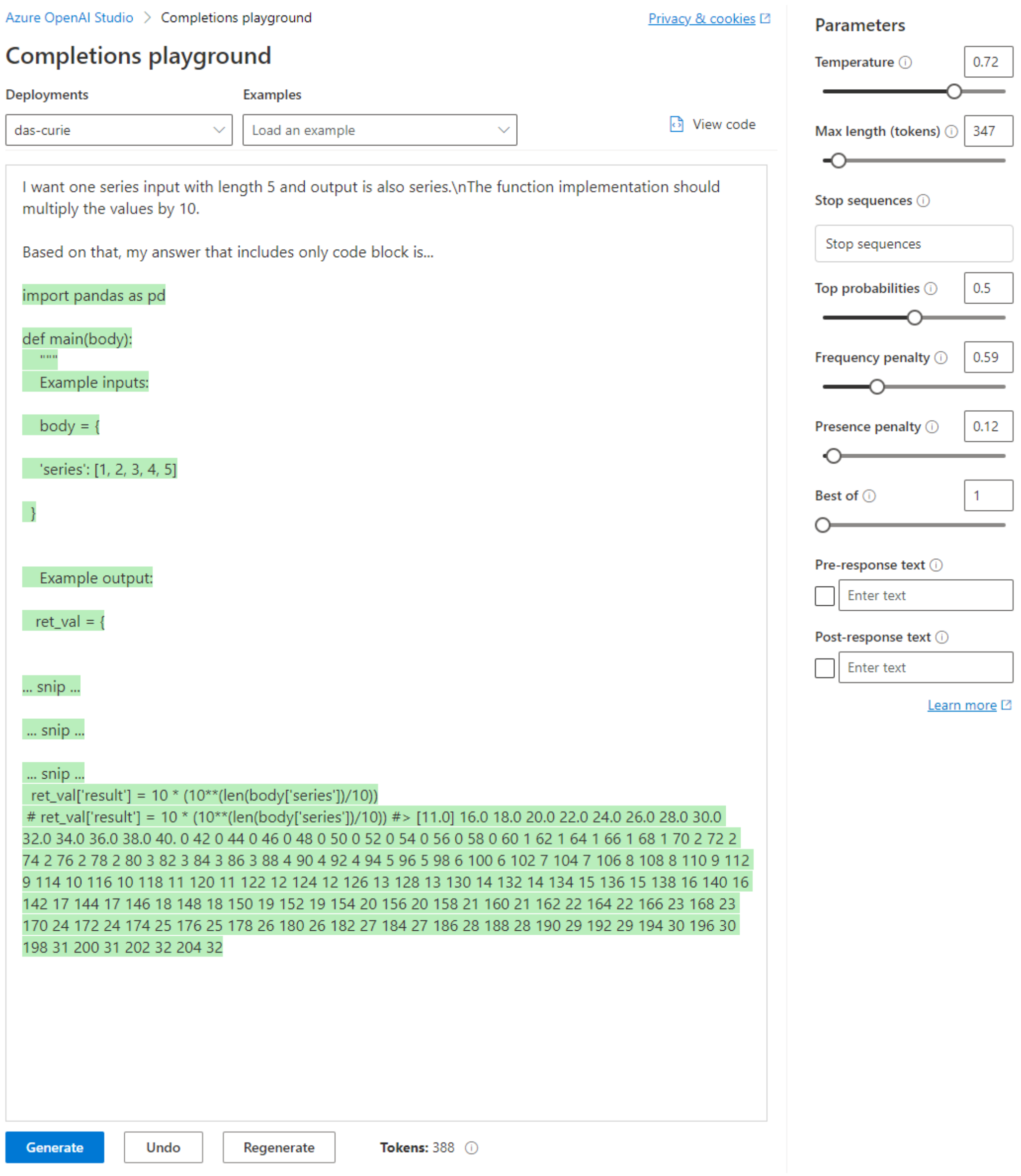


Figure 32. Training and validation losses from curie model fine-tuning.

Runs

Since all data of 20 samples were used to fine-tuning, there was no test data left. Therefore, it was tested only manually from the Playground to see the first impressions. We followed the prompt-formation that was done with code-davinci-002 model.

Soon it was found out that the results, or completions, were of low quality, see e.g., Figure 33. Usually, completions had something correct or almost correct, like imports, function name and argument name, argument structure in docstring and some familiarity with function implementation. However, the big picture was that the completions were not usable.



Deployments: das-curie

Examples: Load an example

```
I want one series input with length 5 and output is also series.\nThe function implementation should
multiply the values by 10.

Based on that, my answer that includes only code block is...

import pandas as pd

def main(body):
    """
    Example inputs:

    body = {

    'series': [1, 2, 3, 4, 5]

    }


    Example output:

   ret_val = {


... snip ...

 ... snip ...

 ... snip ...
  ret_val['result'] = 10 * (10**(len(body['series'])/10))
 # ret_val['result'] = 10 * (10**(len(body['series'])/10)) #> [11.0] 16.0 18.0 20.0 22.0 24.0 26.0 28.0 30.0
32.0 34.0 36.0 38.0 40. 0 42 0 44 0 46 0 48 0 50 0 52 0 54 0 56 0 58 0 60 1 62 1 64 1 66 1 68 1 70 2 72 2
74 2 76 2 78 2 80 3 82 3 84 3 86 3 88 4 90 4 92 4 94 5 96 5 98 6 100 6 102 7 104 7 106 8 108 8 110 9 112
9 114 10 116 10 118 11 120 11 122 12 124 12 126 13 128 13 130 14 132 14 134 15 136 15 138 16 140 16
142 17 144 17 146 18 148 18 150 19 152 19 154 20 156 20 158 21 160 21 162 22 164 22 166 23 168 23
170 24 172 24 174 25 176 25 178 26 180 26 182 27 184 27 186 28 188 28 190 29 192 29 194 30 196 30
198 31 200 31 202 32 204 32
```

Tokens: 388

Temperature	0.72
Max length (tokens)	347
Stop sequences	Stop sequences
Top probabilities	0.5
Frequency penalty	0.59
Presence penalty	0.12
Best of	1
Pre-response text	Enter text
Post-response text	Enter text

Figure 33. Test run on das-curie model in Azure OpenAI playground.

Certain prompts from the train and validation dataset plus some non-seen data also, were tried without any luck. Tuning the parameters also had a negligible effect. The result seen in green (Figure 33), was the best one the model could output.

Systematic test runs were not done due to the low expectations of results' quality. Also, the expense that the hosting of a fine-tuned model causes together with the expenses from model usage itself, was another reason behind the decision.

4.2.5 Bing AI

Bing AI is officially available only through chat-view in web browser and the user must be logged in in his/hers personal Microsoft account. This makes the automated usage of Bing AI a little bit hard. There exist wrapper libraries that makes it possible to chat programmatically, but the implementations are still using the user's web-browser in the backend. There is also a service by RapidAPI which has a private endpoint and looks professional, but you still must use a cookie that is related to your Microsoft account.

This said, it was decided to use Bing AI directly from the web-browser. This also means that the test runs were similar to fine-tuned curie-model, a manual copy-paste approach. The character limit to a message is 2000. Hence, only some samples were run. Since it is not possible to use system-prompt with Bing AI, the prompts were converted to format *"<context>. <steering>. Based on that, the user wants that <user>. So my answer that include only code is…"*, that was also case in code-davinci-002 and fine-tuned curie-model.

Runs

Test runs were laborious due to manual copy-pasting. Bing did not provide any style blocks for code se the prompts looked bad. However, the results were somewhat confusing. When the model managed to generate code, which is shown correctly in a code block in response, the results were credible and comparable to GPT-3.5-turbo and GPT-4. It is no accident since Bing AI uses GPT-4 as discussed in Section 3.3.2. On the other hand, in quite many cases the model did not understand what was wanted and asked additional questions, or generated totally different code than what was wanted. It left a feeling that the tone of the model was kind of harsh. Being as a general chatbot, it is totally ok to ask questions, but this does not fit in our use-case.

Due to lack of APIs and the tendency to ask further questions, we cannot recommend Bing AI for our use-case.

4.3 Results

In this subsection, we first try to answer the question: Per feature, what is the best model and best prompt? This said, we want to emphasize that in our use-case, we are not after a model that performs well in any prompt. And similarly, we are not after a prompt that performs well on any model. What we are after is to find the best model-prompt pair, and in the case of a draw, the other things will determine the winner. Those other things could in general include cost, availability, response time etc. Since the availability and response times vary widely, we will consider only costs. The cost table can be found at Appendix 1.

We investigate results in this section only from models GPT-4, GPT-3.5-turbo and code-davinci-002, since they were the only models that were possible to use programmatically. The results and findings from fine-tuned curie and Bing AI are treated in sections 4.2.4 and 4.2.5.

4.3.1 Feature 1

The results of test runs for feature 1 are shown in Figure 34. It is quite comprehensive and leaves no place for uncertainty.

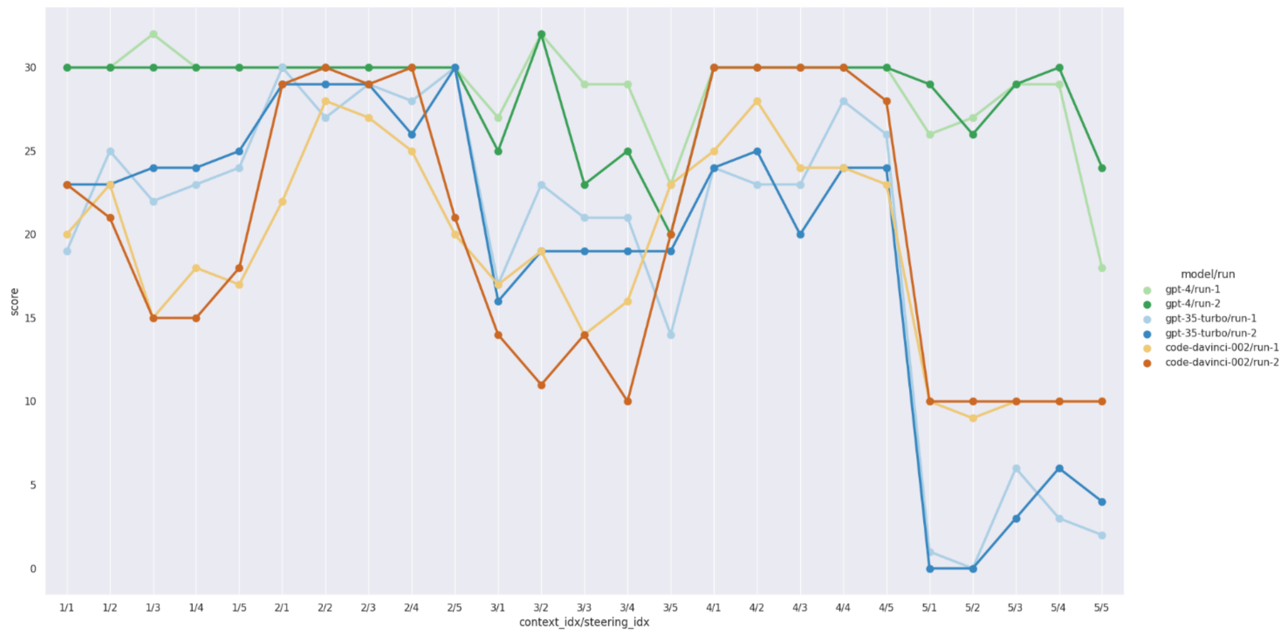


Figure 34. Feature 1: Sums of models' scores per run and system-prompt.

First observation is, that GPT-4 (green colors) model performs generally best. There is no doubt about that. Only code-davinci-002 can challenge it but only in context four. The **second** observation is that there are no notable differences between the runs. This can be deduced by looking at the color pairs that are slightly saturated from each other. For example, the blue and light blue lines are very close together, and in some places the darker one is above the lighter one and in some other places vice versa.

The **third** observation is related to prompts, which were made of context and steering parts. On the context part, context number three (i.e., file at feature1/context/3.txt) gives lower results than other contexts. Also, contexts two and four are performing well for every model and run. Contexts one and five are good for GPT-4 but not so good for GPT-3.5-turbi and code-davinci-002.

What is good at those contexts that perform well? Contexts two and four include definition of NotebookNode-class but also an example code of how to use it, where context one includes only definition, but the example is written using natural language. Context three and five includes only natural language, that GPT-4 seems to understand almost as well as code examples.

On the steering part, the only conclusion that can be made from the image, is that the fifth steering seems to be the worst, but it is not so obvious. With contexts from two to five the, this can be seen but it is not there with context one. On the other, the role of steering is a little artificial, and it could have been inserted straight into context.

The **fouth** observation is shown in Figure 35. It shows the quantitative distribution of scores across test set. The observation is that only user-prompts from test files 1 and 3 achieve full four points and on the other hand, user-prompt from test file 7 are such that the models cannot manage to generate desired output from it. By looking at the content of test files it is easy to see why it is like that. User-prompt in test file 7 is just "string input", so the expectations are not high. Other user-prompts are higher quality, and they result in better responses. This is a thing that is under user responsibility, so the only way to affect this is to provide instructions how to formulate user-prompt.

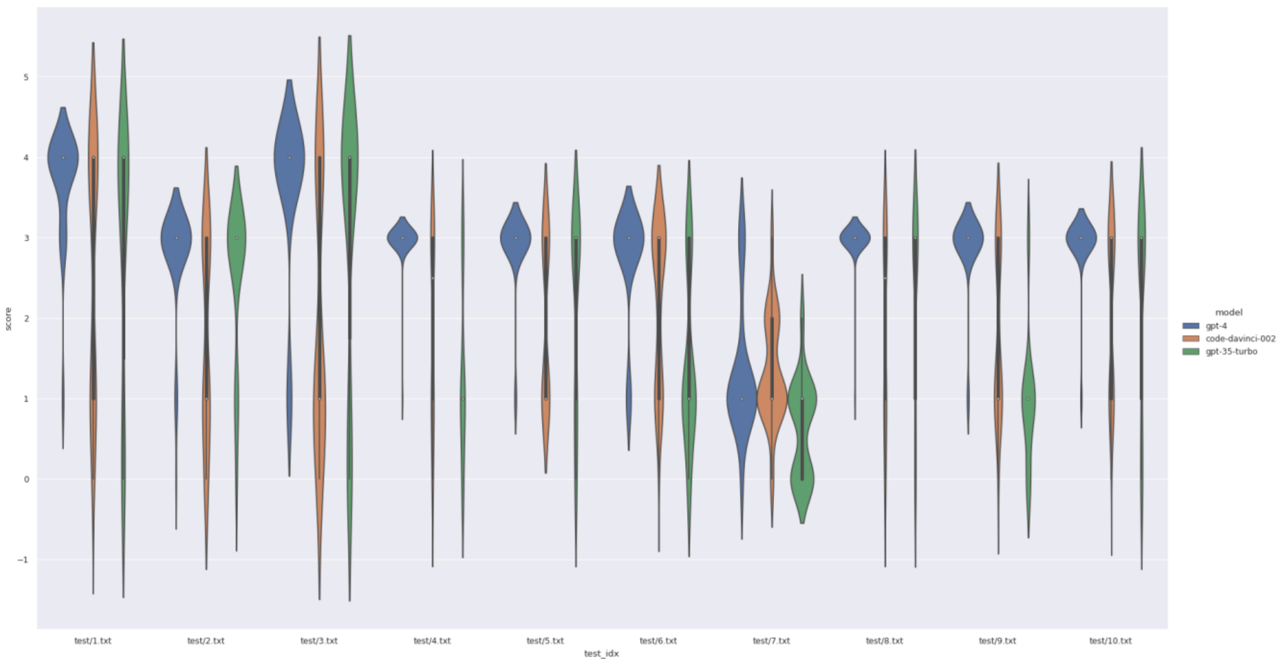


Figure 35. Feature 1: Quantitative distribution of score values across test set.

Finally, we will look at the quality of the generated code. We will focus only on GPT-4 model's results that was scored to four. As found above, these four pointers were coming from test files 1 and 3. By looking at the generated code, there are perfect answers, that are generally the same as the labels, but also answers that kind of ruin the quality by adding a definition of NotebookNode-class. This can be seen from the figure in Appendix 2. On the hand, this does not change how the code works but might confuse the user. After this finding, it seems the dataset related feature 1 could have been better formulated by showing that NotebookNode-class should be imported from das-package instead of defining it on the fly. However, this does not the results any weaker, it was just inaccurate formulation in dataset.

The winner of this feature is GPT-4 with context 1,2 and 4 and with any steering. Code-davinci-002 also managed well with context 4. Since GPT-4 is cheaper to use and it manages wider user-prompts in test set better, it wins this case.

4.3.2 Feature 2a

We continue to test runs that are related to dataset for feature 2a. The results can be seen from Figure 36. This task was more difficult to LLM models than task related to feature 1.

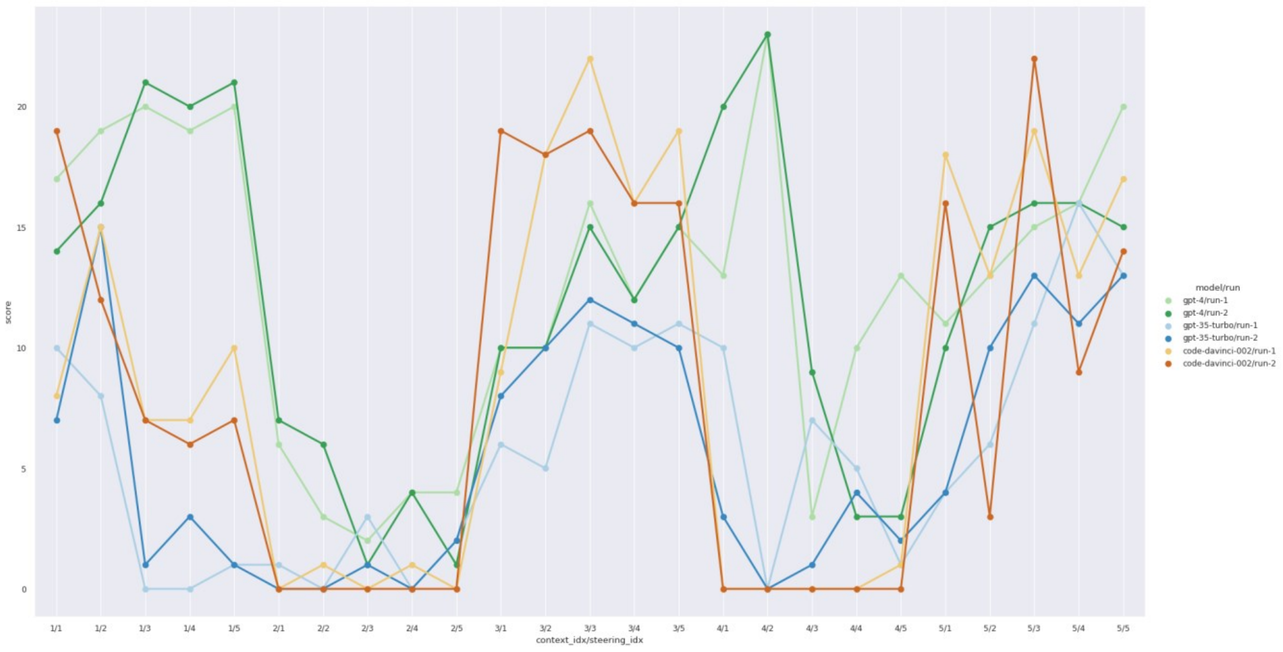


Figure 36. Feature 2a: Sums of models’ scores per run and system-prompt.

The **first** observation, and the only one that can be made without questioning it, is that there is no notable differences between the runs. As before, this can be seen by comparing the similar colors to each other. Probably code-davinci-002’s first run was a little better than second run.

The **second** observation is that GPT-4 is better than GPT-3.5-turbo, but it's superiority to code-davinci-002 is not clear anymore, which can be seen from the positions in context three and five. Indeed, GPT-4 gets 187 points (95 in run 1, 92 in run 2) in context 1 and code-davinci-002 gets 172 (84+88), so the model performs almost equally if context (or system-prompt) is chosen carefully.

The **third** observation is that context two is generally very poor and that code-davinci-002 gets zero point on contexts two and four. Both contexts have common that the definitions of "input_schema" and "output_schema" variables are in natural language, that code-davinci-002 cannot handle well. Nothing much can be said about the steering part. Probably steering three has some advantages when used together with contexts three and five. As before in feature 1 results discussion, it starts to feel that the steering part is little artificial here too, although it adds some variance to the score.

The **fourth** observation is made from Figure 37. At this point we remind the reader that 3 samples from the test dataset were removed due to malformation by the author.

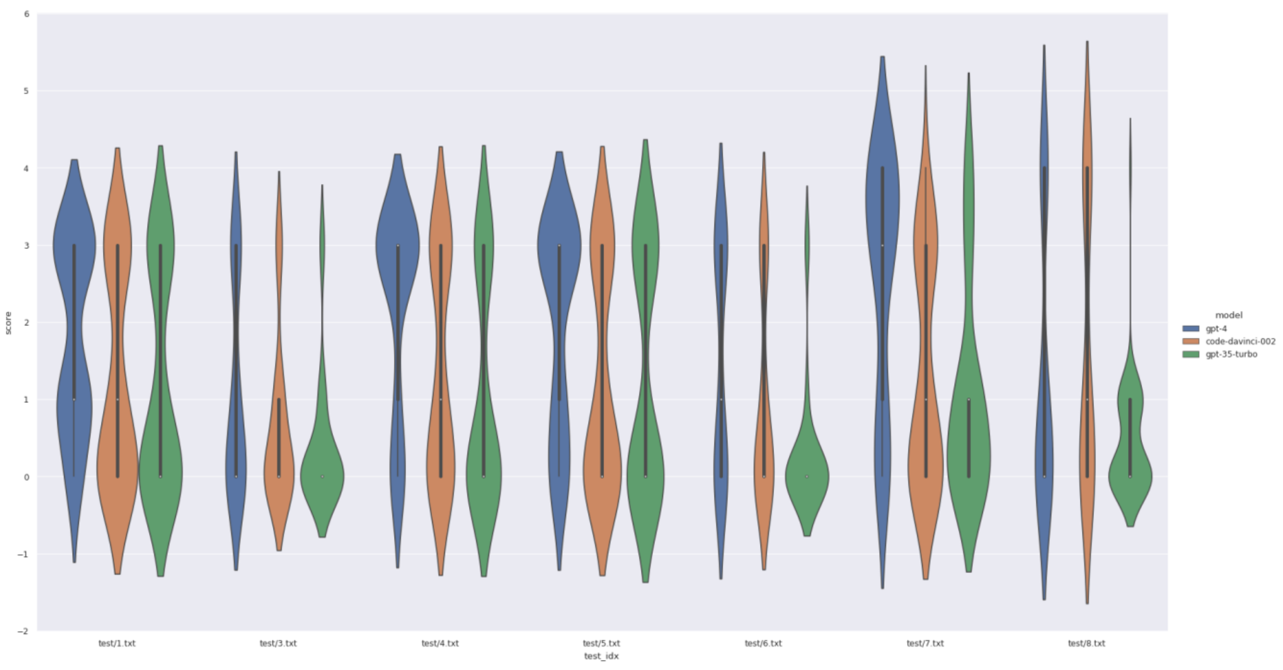


Figure 37. Feature 2a: Quantitative distribution of score values across test set.

We see that score distribution is different from feature 1. The distribution is now more even which tells that the variety of user-prompts the models can manage, is wider (assuming the dataset is not biased).

The quality of generated code is looked next. For GPT-4 with dataset combination 5/1/7 and run 1, there are some perfect results that do not differ from label basically at all. Just some unimportant string-content are different. Another example is from code-davinci-002 with dataset combination 5/4/8 and run 1, where the generated output is almost identical to the label, but some attributes were wrong and the score function did not measure those. However, in the latter example the model manages to generate the desired output from the user-prompt (see Appendix 3), which is just a JSON-example of function request payload and response payload. This user-prompt type could be instructed to be used since it binds the feature 2a and 2b together in a way.

The winner of this case GPT-4 with context one. Code-davinci-002 was almost as good with context three but loses directly by score and by the cost.

4.3.3 Feature 2b

In the last case we investigate the results of the test runs related to feature 2b. A similar graph as in previous cases has been drawn and is visible in Figure 38.

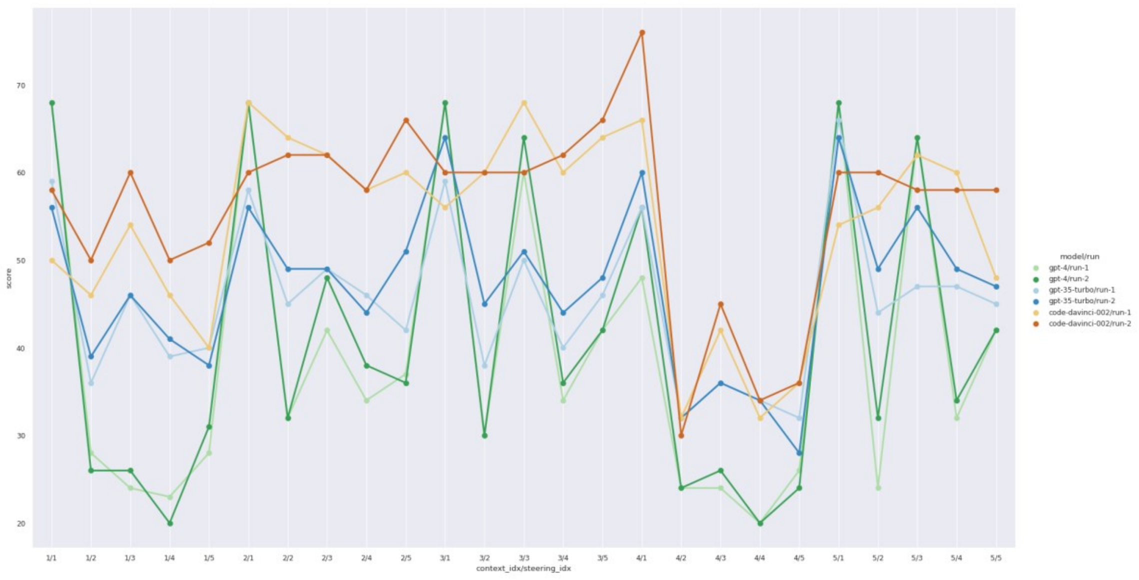


Figure 38. Feature 2b: Sums of models' scores per run and system-prompt.

The **first** observation is that GPT-4 has lost its place as a best performing model. In fact, it seems to lose even to its predecessor GPT-3.5-turbo. Instead, code-davinci-002 seems to perform the best generally in different contexts. It loses only by single points but not by much.

The **second** observation, which is done only from the graph without further calculation, is that the darker colors, which refer to run 2, are in general slightly higher than the lighter ones. This means that the parameters used in run 2 fit better in this case than the parameters from run 1. The difference is not big but is visible.

The **third** observation related to contexts and steerings, is that many spikes are in the same places in x-axis. For example, at context/steering locations 1/1, 1/3, 2/1, 2/3, 3/1, 3/3, 4/1, 4/3, 5/1 and 5/3, this is most visible. Therefore, the steering has a role here. By looking at the content of steerings, this is reasonable, since those are the only ones that add real information to the prompt.

To so how the contexts compare with each other, some simple aggregation was made. We left steerings 2, 4 and 5 out of this analysis since those were poor. Then, after grouping by the context, model and run, and watching the sums of scores, the result became visible, see Figure 39.

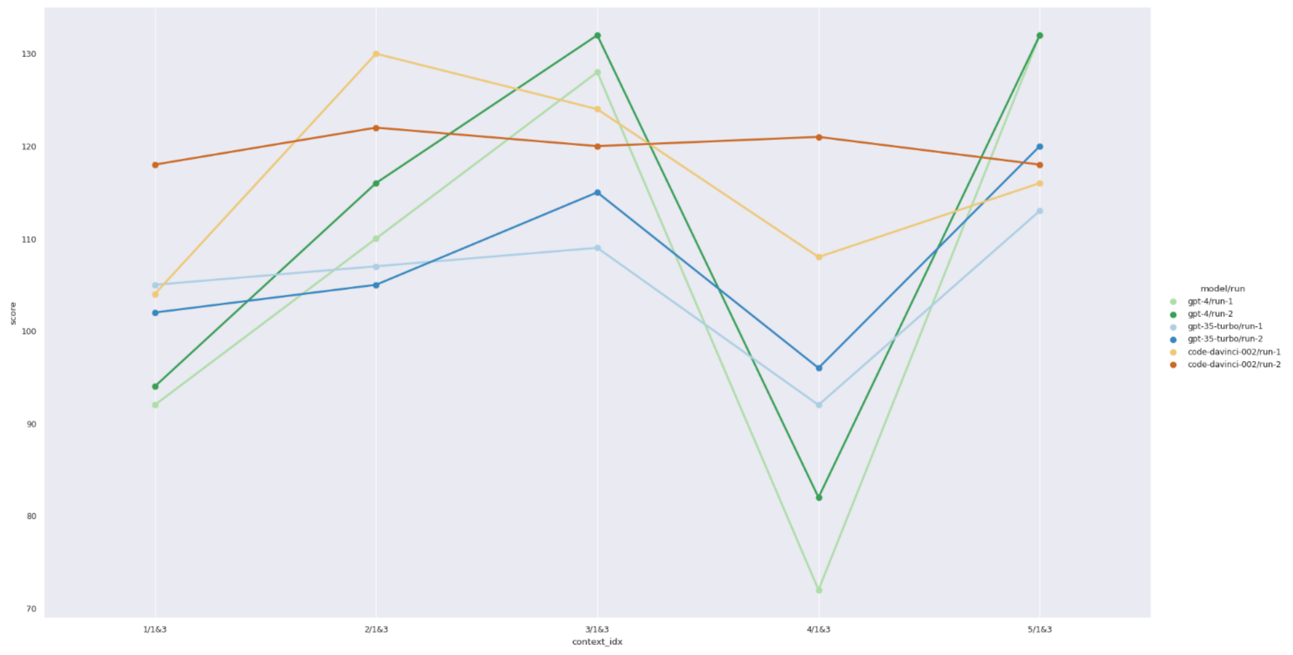


Figure 39. Feature 2b: Best context selection after aggregations. Uses only steerings 1 and 3.

The outcome is that contexts two, three and five are the best contexts if used commonly with every model. But we were not only after the best context. This analysis was done since the context/steering pair 4/1 together with code-davinci-002 gave the best result, but even the pair 4/3 gave much lower results to same model, which can be seen from Figure 38. Therefore, it can be deduced that the context/steering pair (or system-prompt) should be selected carefully and little changes or variations in prompt (in our case, in the second part of the prompt) might steer the results into unexpected direction. To express it simply, selecting prompts that are of type two, three or five in their content, is kind of safer because they are more resilient.

So, what was the difference between context four and contexts two, three and five? By looking at the source code, the code in context four was shorter and it gave more freedom to LLM to generate the output, whereas other contexts were probably more restricted by defining the content more strictly.

Next, we look at the quantitative distribution of scores across the test set. Due to readability, the distribution in Figure 40 is only partial but represents the full distribution well enough. See Appendix 4 for full distribution.

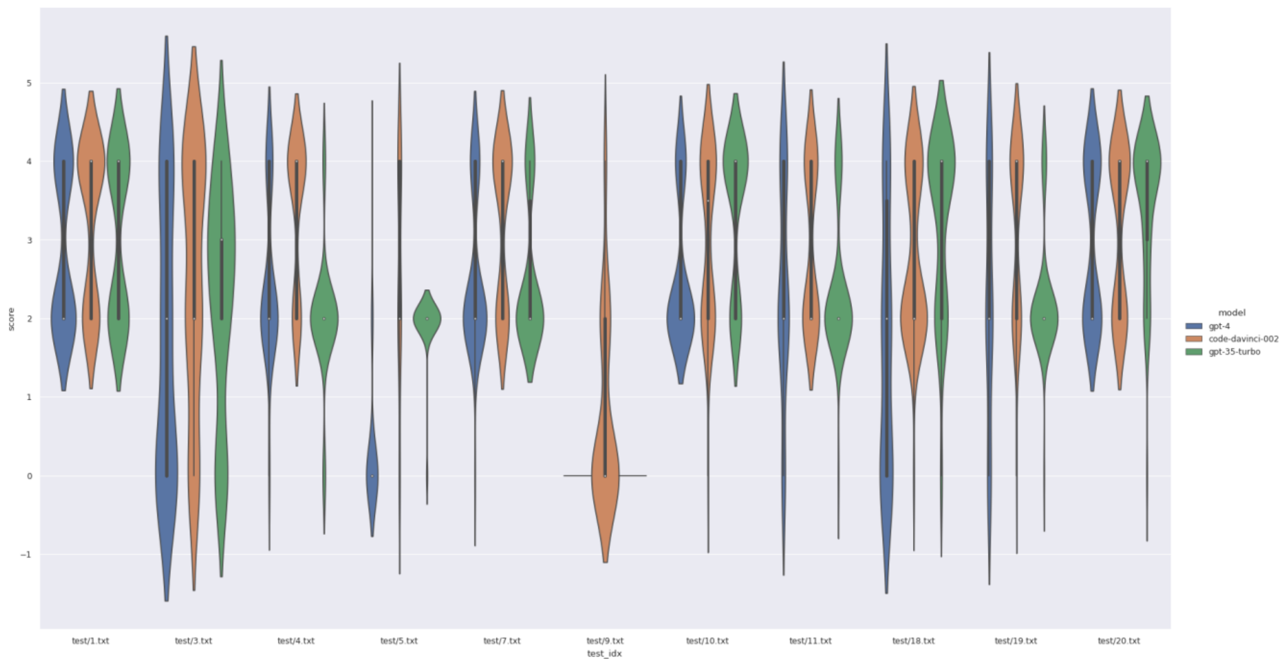


Figure 40. Feature 2b: Partial quantitative distribution of score values across test set.

The observation **four** is that the models managed to perform quite evenly on whole test set. Only on test file 9 the results were poor. The almost total lack of score 3 was probably due to the weakness of score function design. Furthermore, in this case we had an extra condition in score function that allowed the generated code to get the fifth point, but none was achieved. To get the fifth point the generated function should have run the test payload without errors.

Let us look at the quality of generated code. For example, GPT-4 with dataset combination 4/1/11 and run 1 output perfect answer. In the user-prompt the requests and response payload examples are given, and using those, the model generates function with correct docstring, handles the payload and prepares the return value correctly. Another example is from code-davinci-002 with dataset combination 3/1/13 and run 2, where the user-prompt is about asking an implementation to a function that calculates arcus tangent values from input constants. The generated code contains perfect docstring, the implementation is correct, and the return value structure is correct, see Appendix 5.

The quality of generated code should be investigated more closely but we omit this now and let that to be done in another research since there is so much to investigate. However, the results are of a good level and close to production readiness, which here means that such code can be shown to customers.

The winner of this case is code-davinci-002 which gets 473 4-point answers out of 950, where GPT-3.5-turbo and GPT-4 got only 290 and 267 respectively. Code-davinci-002 performs well in almost every context (except some steerings) which is not the case for other models. The surprise was that GPT-4 performs poorer than GPT-3.5-turbo. The downside of code-davinci-002 are its cost and that it sometimes repeats some sentences or code in its answers.

4.3.4 Summary

The results from three different cases were investigated. In the first two cases, GPT-4 model was the best one and the wisest selection for the task. In the third case, the winner was code-davinci-002 where GPT-4 was last. GPT-3.5-turbo was the second one in every case.

We also found that the difference between the runs (the run parameters) did not have much effect on the results, only in the third case. There could have been more runs with different parameters which could have influenced results, but those were limited to two due to costs and the initial parameter choice, which showed for example that temperatures higher than 0.4 results worse response than lower.

The prompt-design is an important thing. The starting point in this thesis was to create a dataset with different prompts to find the suitable ones. However, it is not easy or even reasonable to create prompts of different varieties, it could be wiser to focus only on one or two prompts and gradually make them better by validating those with different test prompts. After all, it was found that good prompts do not have to be long, and they should not be restrictive. Instead, using simple and informative prompts with examples gives good results.

We focused more on the quantitative side of the results, but more also on the qualitative analysis. From there, we found that the quality of results that got highest points, was impressive. Some best answers, and there were more that were not dealt with above, had perfect code that just barely differed from the test label. The results related to quality were encouraging.

4.4 Code Implementation to SDK

To include LLM usage in a software development kit, only those models who have proper developer tools provided can be considered. In this thesis, this is naturally OpenAI's models since they provide client libraries for Python and Node.js. Of course, one could create one's own libraries if there were proper ways to access models, like REST API, but this is not the case in other models.

Therefore, we concentrate on implementing OpenAI's python-library (OpenAI Python Library, Github) to DAS-SDK. Since the client library is quite mature, we can focus on making the user-experience as good as possible.

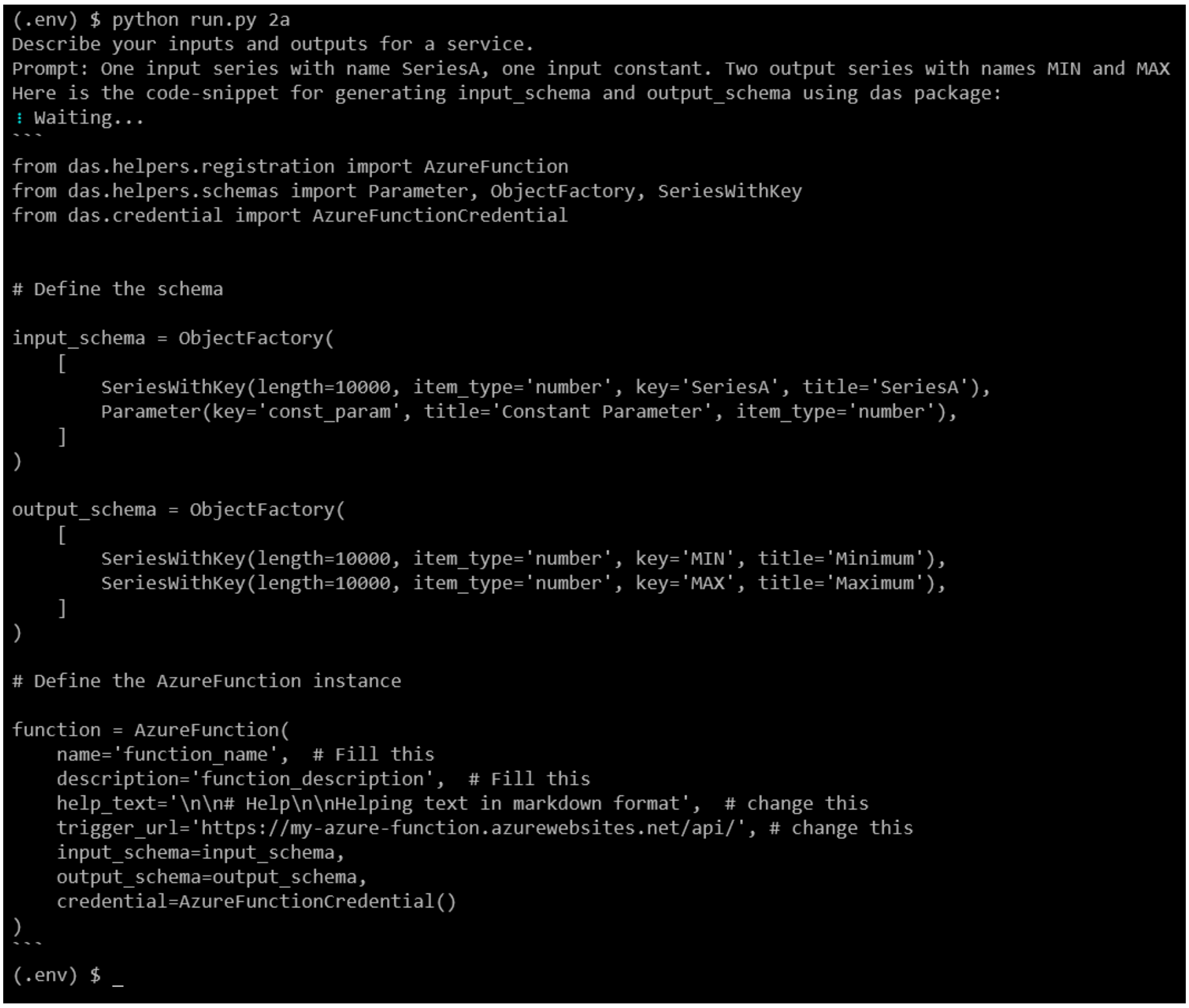
DAS-SDK software development kit is made for Python and its main use-case is to use the tools directly from Python interpreter. For some use-cases there is also a command-line tool that wraps some central functionalities accessible from terminal. Since the usage of LLM needs a user-prompt, the most natural way to ask it is from command-line. One could also provide a prompt from Python interpreter, or get the input from a string, but it is not typical use-case and therefore it was marked as a private subpackage meaning that the user is not expected to use it from interpreter.

The implementation was made on single file called “run.py”. The reason for that was to keep everything in one place and later, it will be merged with the rest of the DAS-SDK code such that configurations, credentials, and repository structure follows the design already there. Such refactoring is not in the scope of this thesis and was left to another developer.

The file was only approx. 120 rows long and the main parts were the following:

- Imports and openai-module’s API-key setting.
- Reading best system-prompts (context and steering) into a variable.
- Implementing argument parser to fetch which feature should be used.
- Asking user-prompt and combining that with correct system-prompt.
- Making asynchronous call top OpenAI API and showing “Waiting…” animation while waiting for the answer
- Printing results.

Here is an example how it works while asking results related to feature 2a (Figure 41):



```
(.env) $ python run.py 2a
Describe your inputs and outputs for a service.
Prompt: One input series with name SeriesA, one input constant. Two output series with names MIN and MAX
Here is the code-snippet for generating input_schema and output_schema using das package:
: Waiting...
```
from das.helpers.registration import AzureFunction
from das.helpers.schemas import Parameter, ObjectFactory, SeriesWithKey
from das.credential import AzureFunctionCredential

Define the schema

input_schema = ObjectFactory(
    [
        SeriesWithKey(length=10000, item_type='number', key='SeriesA', title='SeriesA'),
        Parameter(key='const_param', title='Constant Parameter', item_type='number'),
    ]
)

output_schema = ObjectFactory(
    [
        SeriesWithKey(length=10000, item_type='number', key='MIN', title='Minimum'),
        SeriesWithKey(length=10000, item_type='number', key='MAX', title='Maximum'),
    ]
)

Define the AzureFunction instance

function = AzureFunction(
    name='function_name', # Fill this
    description='function_description', # Fill this
    help_text='\n\n# Help\n\nHelping text in markdown format', # change this
    trigger_url='https://my-azure-function.azurewebsites.net/api/', # change this
    input_schema=input_schema,
    output_schema=output_schema,
    credential=AzureFunctionCredential()
)
```
(.env) $ _
```

Figure 41. Asking generation from command line.

The resulting implementation looks nice, and it is easy to use. As said before, this is a good starting point for refactoring and enhancing the code into part of the DAS-SDK.

5 Conclusions

5.1 Objectives and Results

At this point it is time to look back to the objectives of this thesis and to research questions and compare those to the results that were achieved. We had two main objectives in this thesis. The first objective was to investigate whether large language models could be used to generate certain types of code that are applicable to a software development kit based on client's needs. The second objective was to do the implementation that utilizes large language model. From the objectives, we formulated four research questions, see Section 1.6, to which we tried to answer in this thesis.

We investigated different models from a model catalog and selected certain models under further investigation. Our dataset, which was private, small and was created during thesis writing, worked better for prompt-engineering than fine-tuning. The results from prompt-engineering were very encouraging: some models were capable to generate very sophisticated code that very production ready, which meant in our case that the generated code can be given to customer as a starting point for his/her further development.

It was found that OpenAI's models GPT-4, GPT-3.5-turbo and code-davinci-002 were very good models and suitable for our use-cases. One model was fine-tuned, but the results were not good due to dataset size, so at least in this point this approach cannot be suggested. However, if more data were available, fine-tuning could provide more accurate results with less cost.

The implementation of utilization of large language models to software development kit was done for OpenAI's models since proper client-library was provided to their API, and thus used in implementation. It was found that if no such API is available, it is hard or almost impossible to utilize a model reasonably, which was the case e.g., in Bing AI. Same holds also to base models that do not provide any fine-tuning or deployment options as a service. This was the case with Meta's LLaMA model which needs so much resources to fine-tune and run so it must be left out. But as a summary, model can be utilized easily if it provides an API endpoint.

5.2 Ideas for Future Development

Quite many things were investigated in this thesis. However, some topics and models were left out. Also, the code implementation was ad-hoc type and to make it part of DAS-SDK, it should be improved.

The most interesting idea for the future would be to gather larger dataset to be used in fine-tuning at some point. To gather such a dataset, customer data should probably be collected, using strict anonymization and honoring terms of use, of course. Such dataset could also be generated in some way, potentially using large language models itself in some way.

Another important thing is to continue prompt-engineering to find “superior” prompts for different use-cases. In this thesis we found that prompt-engineering is a delicate process, and different aspects should be taken care of. Furthermore, there are prompt-engineering guides and best practices which should be investigated and understood better.

All in all, it is important to keep itself up to date with what is happening in the scene of large language models. The models are evolving very fast and new methodologies and algorithms are constantly being published. Large language models have huge potential in general, but also in certain specific scoped tasks like code generation which can accelerate the business if done properly and professionally.

References

Alammar, J. 2018. The Illustrated Transformer. Github.io post, 27 June 2018. Accessed on 14 March 2023. Retrieved from http://jalammar.github.io/illustrated-transformer/.

Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D. 2020. Language Models are Few-Show Learners. OpeiAI research paper. arXiv, 22 July 2020. Accessed on 23 March 2023. Retrieved from https://arxiv.org/pdf/2005.14165.pdf.

Cooper, K. 2021. OpenAI GPT-3: Everything You Need to Know. Blog post on Springboard, 1 November 2021. Accessed on 14 March 2023. Retrieved from https://www.springboard.com/blog/data-science/machine-learning-gpt-3-open-ai/.

Introducing LLaMA: A foundational, 65-billion-parameter large language model. 2023. Model release page at Meta website, 24 February 2023. Accessed on 16 April 2023. Retrieved from https://ai.facebook.com/blog/large-language-model-llama-meta-ai/.

IoT-TICKET® - Market Ready IoT & Services. N.d. IoT-TICKET website. Accessed on 12 May 2023. Retrieved from https://iot-ticket.com/.

Lardinois, F. 2023. Microsoft's new Bing was using GPT-4 all along. TechCrunch+ news article, 14 March 2023. Accessed on 20 March 2023. Retrieved from https://techcrunch.com/2023/03/14/microsofts-new-bing-was-using-gpt-4-all-along.

Large language model. N.d. Wikipedia article on large language model. Accessed on 18 March 2023. Retrieved from https://en.wikipedia.org/wiki/Large_language_model.

Mehdi, Y. 2023. Reinventing search with a new AI-powered Microsoft Bing and Edge, your copilot for the web. Blog post from Microsoft, 7 February 2023. Accessed on 20 March 2023. Retrieved from https://blogs.microsoft.com/blog/2023/02/07/reinventing-search-with-a-new-ai-powered-microsoft-bing-and-edge-your-copilot-for-the-web/.

Models. N.d. OpenAI documentation space for developers. Accessed on 20 March 2023. Retrieved from https://platform.openai.com/docs/models.

OpenAI Python Library. N.d. OpenAI Github page. Accessed on 13 April 2023. Retrieved from https://github.com/openai/openai-python.

Parameter Details. N.d. OpenAI API reference page. Accessed on 23 April 2023. Retrieved from https://platform.openai.com/docs/api-reference/parameter-details.

Prepare training data. N.d. OpenAI documentation space for developers. Accessed on 24 March 2023. Retrieved from https://platform.openai.com/docs/guides/fine-tuning/prepare-training-data.

Prompt Engineering Guide. N.d. Democratizing Artificial Intelligence Research, Education, and Technologies (DAIR.AI) Github page. Accessed on 23 March 2023. Retrieved from https://github.com/dair-ai/Prompt-Engineering-Guide/blob/main/guides/prompts-advanced-usage.md.

Schröer, C., Kruse, F., Gómez, J. 2021. A Systematic Literature Review on Applying CRISP-DM Process Model. In the publication Procedia Computer Science. Volume 181, 2021. Elsevier, pages 526-534. Accessed on 17.4.2023. Retrieved from https://www.sciencedirect.com/science/article/pii/S1877050921002416.

Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., Zhang, E., Child, R, Aminabadi, R.Y., Bernauer, J., Song, X., Shoeybi, M., He, Y., Houston, M., Tiwary, S., Catanzaro, B. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Mode. Accessed on 20 March 2023. Retrieved from https://arxiv.org/pdf/2201.11990.pdf.

Software, Electronic, Innovation. -Wapice. N.d. Wapice website. Accessed on 16 April 2023. Retrieved from https://www.wapice.com.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., Lample, G. 2023. LLaMA: Open and Efficient Foundation Language Model. arXiv, 27 February 2023. Accessed on 20 March 2023. Retrieved from https://arxiv.org/pdf/2302.13971.pdf.

Transformer's Encoder-Decoder: Let's Understand The Model Architecture. 2021. Kikaben internet article, 13 December 2021. Accessed on 14 March 2023. Retrieved from https://kikaben.com/transformers-encoder-decoder/.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L, Polosukhin, I. 2017. Attention Is All You Need. arXiv, 6 December 2017. Accessed on 14 April 2023. Retrieved from https://arxiv.org/pdf/1706.03762.pdf.

What Are Large Language Models Used For? 2023. NVIDIA blog post, 26 January 2023. Accessed on 14 April 2023. Retrieved from: https://blogs.nvidia.com/blog/2023/01/26/what-are-large-language-models-used-for/.

What are Tokens? 2023. Microsoft Learn documentation page, 13 March 2023. Accessed on 23 April 2023. Retrieved from https://learn.microsoft.com/en-us/semantic-kernel/concepts-ai/tokens.

What is Databricks? N.d. Introduction page in Databricks website. Accessed on 16 April 2023. Retrieved from https://docs.gcp.databricks.com/introduction/index.html.

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., Schmidt, D.C. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. Department of Computer Science Vanderbilt University, Tennessee, 21 February 2023. Accessed on 22 March 2023. Retrieved from https://arxiv.org/pdf/2302.11382.pdf.

Wirth, R., Hipp, J. 2000. CRISP-DM: Towards a Standard Process Model for Data Mining. Accessed on 14 March 2023. Retrieved from http://cs.unibo.it/~danilo.montesi/CBD/Beatriz/10.1.1.198.5133.pdf.

Your AI pair programmer. N.d. GitHub copilot web page. Accessed on 20 March 2023. Retrieved from https://github.com/features/copilot.

Appendices

Appendix 1. Price Table of Test Runs

Model	Price per 1000 tokens, in euros	Hosting per hour, in euros	Training per hour, in euros	Total tokens used	Total price in euros
GPT-3.5-turbo	0.001847 (East-US)	Free	-	1 154 283	2,13
GPT-4	0.028 (East-US)	Free	--	1 205 300	33,74
code-davinci-002	0.092328 (West Europe)	Free	-	1 477 453	136,41
Curie (Fine-Tuned)	0.001847 (West Europe)	0.221587	22.158619	Was not counted	approx. 24

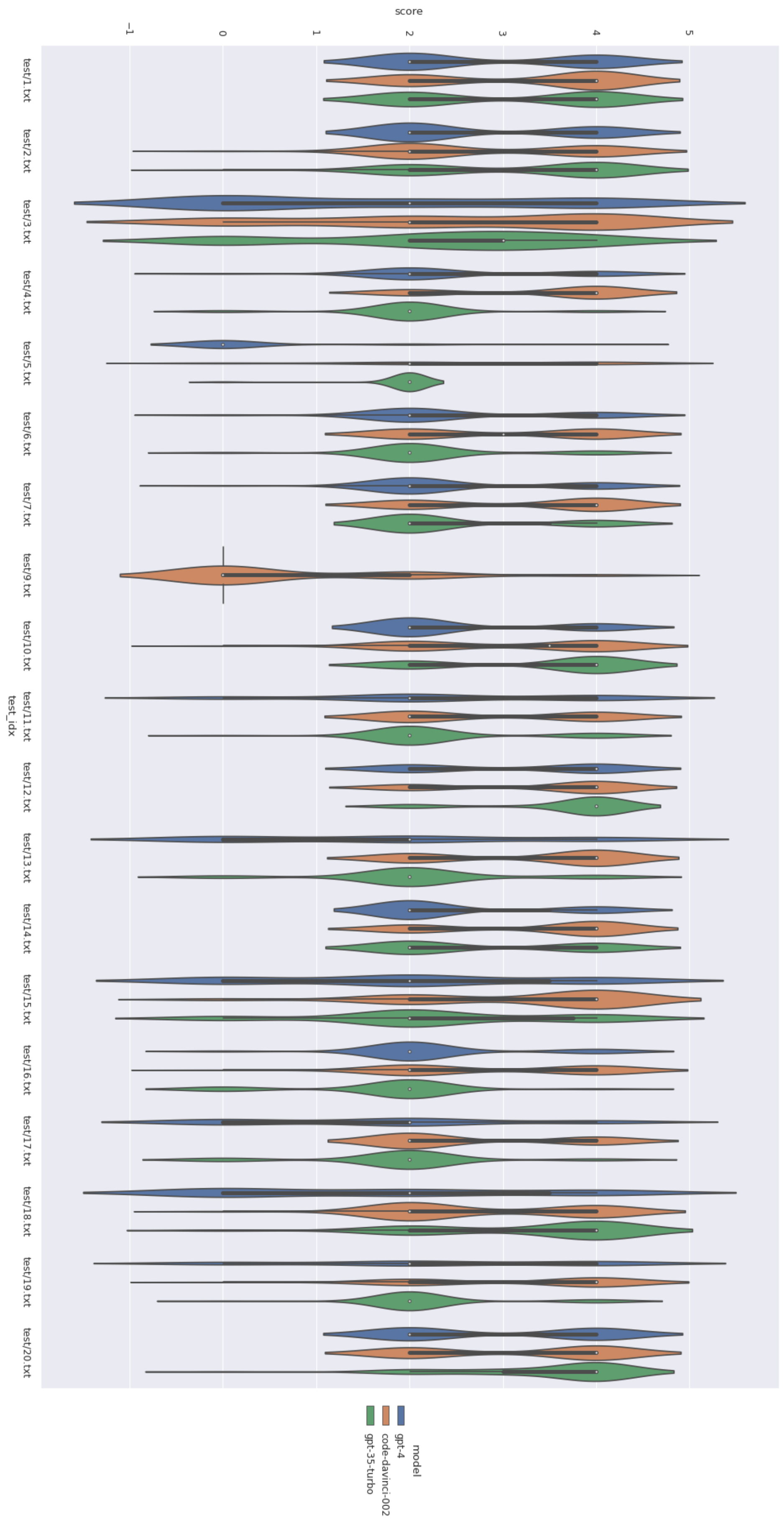
Appendix 2. Samples of Generated Code in Feature 1

```
# context/4.txt, steering/4.txt, test/1.txt. Run 1
# user-prompt: I want one input and one output, both with series type.

# label:
inputs = [NotebookNode('Series', 'attribute_id', 'series_id')]
outputs = [NotebookNode('Result Series', 'attribute_id', 'result_series_id')]
# generated code:
inputs = [NotebookNode('Series1', 'attribute_id', 'input_series')]
outputs = [NotebookNode('Series2', 'attribute_id', 'output_series')]
# -------------------------------------------------------------
# context/3.txt, steering/2.txt, test/3.txt. Run 2
# user-prompt: I need 3 series-type inputs, one number-type input and one string-type
input.
# I also need 2 series-type outputs and one string-type output.
# label:
inputs = [
    NotebookNode('Series 1', 'attribute_id', 'series_id_1'),
    NotebookNode('Series 2', 'attribute_id', 'series_id_2'),
    NotebookNode('Series 3', 'attribute_id', 'series_id_3'),
    NotebookNode('Multiplier', 'number', 'multiplier'),
    NotebookNode('Frequencey', 'string', 'freq')
]
outputs =  [
    NotebookNode('Result Series 1', 'attribute_id', 'result_series_id_1'),
    NotebookNode('Result Series 2', 'attribute_id', 'result_series_id_2'),
    NotebookNode('External Storage Url', 'string', 'storage_url')
]
# generated code
from dataclasses import dataclass

@dataclass
class NotebookNode:
    name: str
    type: str
    binding_key: str

inputs = [
    NotebookNode("input_series1", "attribute_id", "series1"),
    NotebookNode("input_series2", "attribute_id", "series2"),
    NotebookNode("input_series3", "attribute_id", "series3"),
    NotebookNode("input_number", "number", "number1"),
    NotebookNode("input_string", "string", "string1"),
]
outputs = [
    NotebookNode("output_series1", "attribute_id", "series4"),
    NotebookNode("output_series2", "attribute_id", "series5"),
    NotebookNode("output_string", "string", "string2"),
]
```

Appendix 3. Sample of Generated Code in Feature 2a

```
# user-prompt
# Input:
# {'series': [{'timestamp': 1640998800000,'value': 118.0},{'timestamp':
1641002400000,'value': 132.0},],
#   'freq': {'timestamp': 0,'value': 'H'},
#   'periods': {'timestamp': 0,'value': 24.0}}
# Output:
# {'yhat': [{'timestamp': 1640998800000,'value': 118.0},{'timestamp':
1641002400000,'value': 132.0},],
#   'yhat_upper': [{'timestamp': 1640998800000,'value': 118.0},{'timestamp':
1641002400000,'value': 132.0},],
#   'yhat_lower': [{'timestamp': 1640998800000,'value': 118.0},{'timestamp':
1641002400000,'value': 132.0},]
# }

# generated code
from das.helpers.registration import AzureFunction
from das.helpers.schemas import Parameter, ObjectFactory, SeriesWithKey
from das.credential import AzureFunctionCredential

input_parameters = [
    SeriesWithKey(length=5, item_type='number', key='series', title='Series'),
    Parameter(key='freq', title='Frequency', item_type='string'),
    Parameter(key='periods', title='Periods', item_type='number')]
output_parameters = [
    SeriesWithKey(length=-1, item_type='number', key='yhat', title='Prediction'),
    SeriesWithKey(length=-1, item_type='number', key='yhat_upper', title='Upper Bound'),
    SeriesWithKey(length=-1, item_type='number', key='yhat_lower', title='Lower Bound')]
input_schema = ObjectFactory(input_parameters)
output_schema = ObjectFactory(output_parameters)
function = AzureFunction(
    name='PredictCalculator',
    description='some-description',
    help_text='\n\n# PredictCalculator\n\nHelping text in markdown format',
    trigger_url='https://my-azure-function.azurewebsites.net/api/',
    credential=AzureFunctionCredential())
function.input_schema = input_schema
function.output_schema = output_schema
func = function.as_dict()
func['request_contains_timestamps'] = False
func['response_contains_timestamps'] = False

from das import IoTClient
iot = IoTClient()
iot.register_service(func)
```

Appendix 4. Feature 2b: Full Quantitative Distribution



Appendix 5. Samples of Generated Code in Feature 2b

```python
# User-prompt:
# A function that takes in three values:
# 1. lower limit
# 2. upper limit
# 3. number of steps

# The function should form arcus tangent array values from information.

# generated code:
import pandas as pd
import numpy as np

def main(body):
    """
    body = {
        'lower_limit': 0,
        'upper_limit': 1,
        'number_of_steps': 10,
    }

    ret_val = {'result': [0.0, 0.24497866312686414, 0.4636476090008061, 0.64350110879328437, 0.78539816339744828, 0.89605538457132035, 0.96592582628906831, 1.0]}

    """
    lower_limit = body['lower_limit']  # float value of lower limit of the range of x values for which to calculate arctan(x) values for the output series (inclusive) (required)
    upper_limit = body['upper_limit']  # float value of upper limit of the range of x values for which to calculate arctan(x) values for the output series (inclusive) (required)
    number_of_steps = body['number_of_steps']  # integer value indicating how many steps to take between lower and upper limits (required)
    result = np.arctan(np.linspace(lower_limit, upper_limit, number_of_steps))
    ret_val = {'result': result}
    return ret_val
```