Scan Optimization With AI

Diogo Antunes
(nrº 21144, regime diurno)

Orientação de
Óscar Ribeiro

Licenciatura em Engenharia em Sistemas Informáticos
Escola Superior de Tecnologia
Instituto Politécnico do Cávado e do Ave

Identificação do aluno

Diogo Antunes
Aluno número 21144, regime diurno
Licenciatura em Engenharia em Sistemas Informáticos

Orientação

Óscar Ribeiro

Informação sobre o Estágio

Checkmarx
Braga, Portugal
Eng. João Cruz, Eng. Josué Rodrigues, Eng. Nuno Oliveira

Abstract

As digital solutions become integral to organizations, securing software applications is paramount. The report explores the integration of AI-driven query optimization techniques within a SAST framework, specifically at Checkmarx, a leader in software security solutions, focusing on the modularity of the AI model and future directions. The project entailed developing a modular AI solution for sequence classification to enhance the efficiency of SAST tools by filtering unnecessary queries. Key contributions include the design of a versatile sequence classification model, the creation of a flexible framework for future use, and possible future improvements in the speed and accuracy of Checkmarx's SAST tool.

The conclusion highlights the successful implementation of the AI solution, personal growth during the internship, and future work directions, emphasizing the transformative potential of AI in cybersecurity and other fields.

Acknowledgments

The completion of this work would not have been possible without the support and contributions of several individuals and organizations, to whom I express my deep gratitude.

First and foremost, I would like to thank my supervisors, João Cruz, Josué Rodrigues, and Nuno Oliveira, for their patient guidance, valuable insights, and continuous support throughout this project. Their expertise was crucial to the development of this work.

I am deeply grateful to my advisor, Óscar Ribeiro, for their exceptional guidance, unwavering support, and invaluable mentorship throughout every stage of this project. Their expertise, insightful feedback, and encouragement were instrumental in shaping this work and my academic growth.

To my family, for their unconditional love, constant encouragement, and understanding during the long hours devoted to this project. Without their support, this work would not have been possible.

I also extend my thanks to my colleagues and friends who shared ideas, discussed issues, and offered constructive suggestions along the way.

To IPCA, for the support provided through the internship, which enabled the completion of this project.

Last but not least, special appreciation goes to Checkmarx for their support and collaboration throughout this project. Their support, technical assistance, and funding, significantly enhanced the quality and scope of this work.

To all of you, my heartfelt thanks.

Contents

List of Figures

List of Tables

Code Listings

1. Introduction

In Computer Systems and Computer Science, the demand for cybersecurity measures is more critical than ever. As organizations and enterprises increasingly depend on digital solutions, ensuring the security of software applications through methodologies like Static Application Security Testing (SAST) is paramount. This introduction lays the groundwork for understanding the intricate challenges and technological needs of SAST, particularly when enhanced with artificial intelligence technologies for improved efficiency.

SAST tools, which analyze source code, byte code, or binary code for security flaws by examining the application ”from the inside out,” are pivotal in the early stages of the software development lifecycle. By enabling developers to detect and rectify security flaws early, these tools not only enhance the security posture of applications but also significantly reduce the cost and effort associated with addressing security issues in later stages.

However, the efficiency and effectiveness of SAST tools are contingent upon their ability to manage and process large datasets swiftly and accurately, which brings forth the challenge of optimizing these systems. Query optimization, particularly through the use of AI techniques, emerges as a promising solution to enhance the performance of SAST tools. AI-driven query optimization can potentially revolutionize how SAST tools handle data, making them faster and more efficient, thereby aligning with the increasing pace of software development cycles.

1.1 Objectives

The primary objective of this report is to explore and demonstrate the integration of AI-driven query optimization techniques within a SAST framework. Specifically, the goal is to develop a sequence classification solution, designed to filter unnecessary queries.

- Gather and label data.
- Implement a solution for Sequence Classification.
- Optimize checkmarx’s query selection with project type classification.

The optimization of checkmarx’s query selection was not considered to be the intern’s responsibility, only the sequence classification solution and data gathering.

1.2 Context

This work was conducted during an internship at Checkmarx, a leader in providing software security solutions for enterprise-level applications. Checkmarx specializes in static application security testing, focusing on identifying security vulnerabilities in software without executing the program. The project is part of a broader initiative to incorporate AI technologies into Checkmarx's existing SAST solutions to address the growing complexity and volume of data in software development.

1.2.1 Implementation Overview

During this internship, a modular and generic Artificial Intelligence (AI) solution was designed and implemented for sequence classification, enabling its application across various domains. The models were trained from scratch and fine-tuned to achieve optimal performance.

The project offers a modular framework specifically designed for sequence classification. It is intended for developers and researchers who require a flexible approach to model training and deployment.

With minimal effort, an engineer can re-use this code for another problem just by changing the following data/components:

- **Data Collection**: Gather and label datasets for training.
- **Tokenization Engineering**: Re-engineer tokenization strategies if necessary.

The project provides comprehensive tools to:

- **Train a Tokenizer**: Train a tokenizer with Byte Pair Encoding.
- **Pretrain a Model**: Train a model with Masked Language Modeling.
- **Finetune a Model**: Apply finetuning methods to tailor the model to specific tasks and datasets.
- **Evaluate the Model**: Evaluate model's performance.
- **Deploy the Model**: Serve the model with an API.

This project is tailored for engineers who require detailed control over sequence classification tasks.

1.3 Document Structure

The structure of this report is organized as follows:

1. **Introduction**: Provides an overview of the project's context, objectives, and the importance of integrating AI in SAST tools.

2. **Analysis of the Problem**: Discusses how to address the problem and possible solutions.

3. **System Analysis and Modeling**: Details the design and architecture of the proposed sequence classification transformer, including helper diagrams, and some technical concepts for context.

4. **Implementation Details**: Describes the implementation process, the technologies chosen, and the rationale behind these choices. It also covers the detailed implementation process of the AI model.

5. **Data Collection**: Details how how data was collected for the pretrain stage which requires a vast amount of information.

6. **Deployment Details**: Outlines the comprehensive hosting process and infrastructure build to support the system.

7. **Analysis of Results**: Evaluates the effectiveness of the implemented solution through a test dataset. Various training statistics are also presented.

8. **Future Work and Considerations**: Details areas for future improvement and potential enhancements. This chapter suggests further optimizations that could be implemented to enhance the AI model capabilities. An overall analysis of model complexity and dataset size is also detailed.

9. **Conclusion**: Summarizes the findings, discusses potential and reflects on the internship experience and the impact of the project within Checkmarx.

2. Analysis of the Problem

This chapter discusses problem analysis and the primary technologies used to understand and classify sequences in NLP. As data processing needs increase, particularly with the complexity of projects and code development, it is vital to enhance and refine advanced models capable of handling various data types beyond traditional text.

The ”State of the Art” section highlights the innovative features of Transformers that make them superior in sequence classification tasks. Their capability for parallel processing, along with the self-attention mechanism, enables efficient and scalable handling of sequential data, making them ideal for a broad range of NLP applications.

The ”Classification Type” section outlines different Artificial Intelligence (AI) classification methods, emphasizing the adaptability of these models to various classification schemes—binary, multi-class, and multi-label. This flexibility is essential for tackling real-world problems where data often do not fit neatly into single categories.

Furthermore, the ”Solution by Analyzing Code” section provides a practical application of transformers, using CodeBERT to analyze and categorize code snippets. This application demonstrates the efficacy of transformers in specialized fields such as software development, enhancing classification accuracy and efficiency.

Finally, the ”Solution by Analyzing Project Structures” section explains how major issues were addressed, presenting a comprehensive solution for sequence classification by engineering tokenization, model pretraining, and fine-tuning models.

2.1 The State of the Art

Transformers are considered state of the art for Sequence Classification primarily due to their unique architecture and training methods, which confer several advantages:

- Self-Attention Mechanism
- Parallel Processing
- Scalability
- Flexibility
- Transfer Learning

These features make transformers highly effective for sequence classification and a range of other NLP tasks, setting them apart as the current state-of-the-art technology in the field. These advantages will become more clear as the reader progresses towards the end of the report.

2.2 Classification Type

In this section, the three major types of classification problems in AI are detailed. A visualization is provided in Figure 2.1.

- **Binary** - Binary classification involves categorizing the data into one of two distinct groups.
- **Multi-class** - In multi-class classification, each sample is assigned to one of three or more classes. The goal is to predict a single class label from several possible categories.
- **Multi-label** - Multi-label classification differs in that each sample can be assigned multiple labels from a set of possible categories. This type of classification is useful when the categories are not mutually exclusive.

For the specific task of classifying projects, multi-label classification is used. A project can have simultaneous labels at the same time.

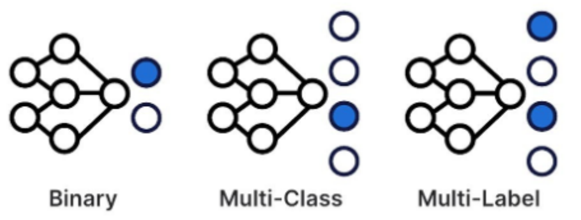


Figure 2.1: Classification Problem Types

2.3 A Solution By Analyzing Code

The described process involves using a Sequence Classification approach to categorize code snippets as backend, frontend, mobile, etc. This is achieved using a transformer-based model, specifically CodeBERT, from the Hugging Face library. Each step in the process is detailed below:

2.3.1 Overview

1 - Load a Base Model

CodeBERT: A Transformer model trained specifically on programming languages, making it particularly suitable for understanding and classifying code. CodeBERT, available from Hugging Face's model hub, contains approximately 140 million parameters, providing a solid foundation for the classification task.

Purpose: The utilization of a pre-trained model such as CodeBERT allows for the leveraging of learned representations of code syntax and semantics, which can be Fine-tuning for specific tasks such as distinguishing between backend and frontend code.

2 - Integrate a Linear Layer (Fully-Connected)

BertForSequenceClassification: This architecture from Hugging Face includes a pre-integrated linear layer situated on top of the transformer outputs, tailored specifically for Sequence Classification tasks.

Simplification: The pre-integration of the necessary linear layer for classification enables the skipping of manual layer addition. This model configuration is newly initialized and primed for Fine-tuning on specific datasets.

3 - Build Annotated Datasets

Types of Data: Three types of datasets are required: training, testing, and validation. Each dataset must be annotated to specify whether a code snippet is related to backend, frontend, mobile, etc.

Purpose: These annotated datasets serve as the ground truth labels necessary for training the model. The model learns from the training data, its performance is assessed on the validation data, and its generalization capabilities are evaluated on the test data.

4 - Tokenize Input

Tokenizer: The tokenizer corresponding to CodeBERT should be employed. This tokenizer is designed to convert code snippets into a format processable by the model, namely by turning raw text into token IDs.

Application: It is crucial to Tokenization all input data of the datasets, with the exception of the labels, which must remain unchanged to ensure the model can be effectively trained to predict them.

5 - Implement Training Epochs

Epochs: This phase involves setting the number of epochs for which the model will be trained. An epoch is defined as a complete pass over the entire training dataset.

Process: With each epoch, the model adjusts its weights to minimize the discrepancies between its predictions and the actual labels. Optimizing the number of epochs is essential to enhance performance and prevent overfitting.

2.3.2 The Big Problem?

This solution was initially employed during the internship, where it quickly became apparent that the inherent limitations posed significant challenges.

The constraint on context length, set at 512 tokens, initially discouraged further use of this approach for larger datasets. Despite achieving excellent results with relatively small datasets, the scalability of the solution was compromised when attempting to handle more substantial projects. This limitation highlighted the need for adaptations or alternative methods capable of managing larger volumes of data without sacrificing efficiency.

2.4 A Solution By Analyzing Project Structures

A similar approach to the one used for analyzing code snippets can be employed to analyze project structures. The steps involved in this process would essentially mirror those used in the Sequence Classification of code:

- Loading a base model suitable for analyzing project structures.
- Integrating a linear classification layer if not already present.
- Building annotated datasets specific to project structures.
- Tokenization the input data to suit the chosen model.
- Implementing training epochs to Fine-tuning the model on the new data.

However, this approach encounters a significant hurdle: unlike natural language, project structures do not inherently align with the methodologies used in NLP. The optimized models currently available are predominantly tailored for text-based NLP, not for structural or non-textual data interpretation. This misalignment necessitates substantial adaptations:

Tokenization Engineering: Adapting Tokenization processes to effectively handle the unique elements of project structures is crucial. Traditional NLP tokenizers are designed to process textual data, which means new tokenization strategies must be developed to parse and interpret the hierarchical and non-linear formats typical of project structures.

Model Base Pretraining: Since existing models are optimized for NLP tasks, there is a need for the development and pretraining of new models that are specifically tuned to the nuances of project structure analysis. This involves not just adjusting the model architecture but also retraining it from the ground up with datasets that accurately reflect the structural data it will encounter.

These challenges underscore the need for innovative solutions that extend beyond the current capabilities of NLP technologies, pushing towards a more flexible and inclusive understanding of different data types within AI models.

This report focuses primarily on the solution by analyzing project structures (as input to the transformer model).

Involving advanced Tokenization engineering, model pretraining, and Fine-tuning. A key feature of this approach is the use of the Longformer model, which handles 4,096 tokens per context as defined in the config. This capability enables the completion of inference requests in just one iteration, improving both time efficiency and resource utilization.

Performance metrics are particularly noteworthy: inference operations on a Central Processing Unit (CPU) take approximately 400 milliseconds, while a Graphics Processing Unit (GPU) with 38 TeraFLOPS, a measure of a computer's speed and can be understood as a trillion floating-point operations per second (TFLOPS) (fp32) of processing power can accomplish the same task in about 20 milliseconds. These results highlight significant improvements in model efficiency compared to analyzing by code samples which could take hours to complete.

3. System Modeling

This chapter delves into the architecture and functioning of transformer models, focusing on their application in processing and understanding complex data patterns for sequence classification. An overview of the model training process is present in figure 3.1.

3.1 High Level Overview

The process of training a language model involves three main stages, each critical to developing an effective model.

1. **Tokenizer Engineering**
   - *Tokenizer Training*: Develop a tokenizer capable of breaking text into manageable units (tokens) that the model can understand.
   - *Tokenizer Component*: Implement the tokenizer as a reusable component that converts raw text into a sequence of tokens.
2. **Pretrain**
   - *Input Tokenization*: Use the tokenizer to convert input data into token sequences.
   - *Masked Language Modeling*: Train the model to predict missing tokens in a sequence to learn contextual relationships between tokens.
   - *Training Epochs*: Run multiple cycles of training to allow the model to iteratively learn from a vast amount of text data.
   - *Base Model*: Establish a base version of the model, which understands general language but is not yet specialized for any specific task.
3. **Finetune**
   - *DataSets Loading*: Load task-specific datasets that will be used to adapt the base model to particular needs.
   - *Input Tokenization*: Tokenize the new datasets using the same tokenizer.
   - *Training Epochs*: Continue training the model on the specific datasets to optimize its performance for particular tasks.
   - *Finetuned Model*: Achieve a version of the model that is specifically tuned to perform well on its intended tasks.

Training a language model typically involves developing a tokenizer, using it to convert text into tokens, and then training the model on these tokens through several epochs to learn general and specific linguistic patterns. This process starts with pretraining on general data for broad understanding and is followed by finetuning on specific datasets for task-specific performance.

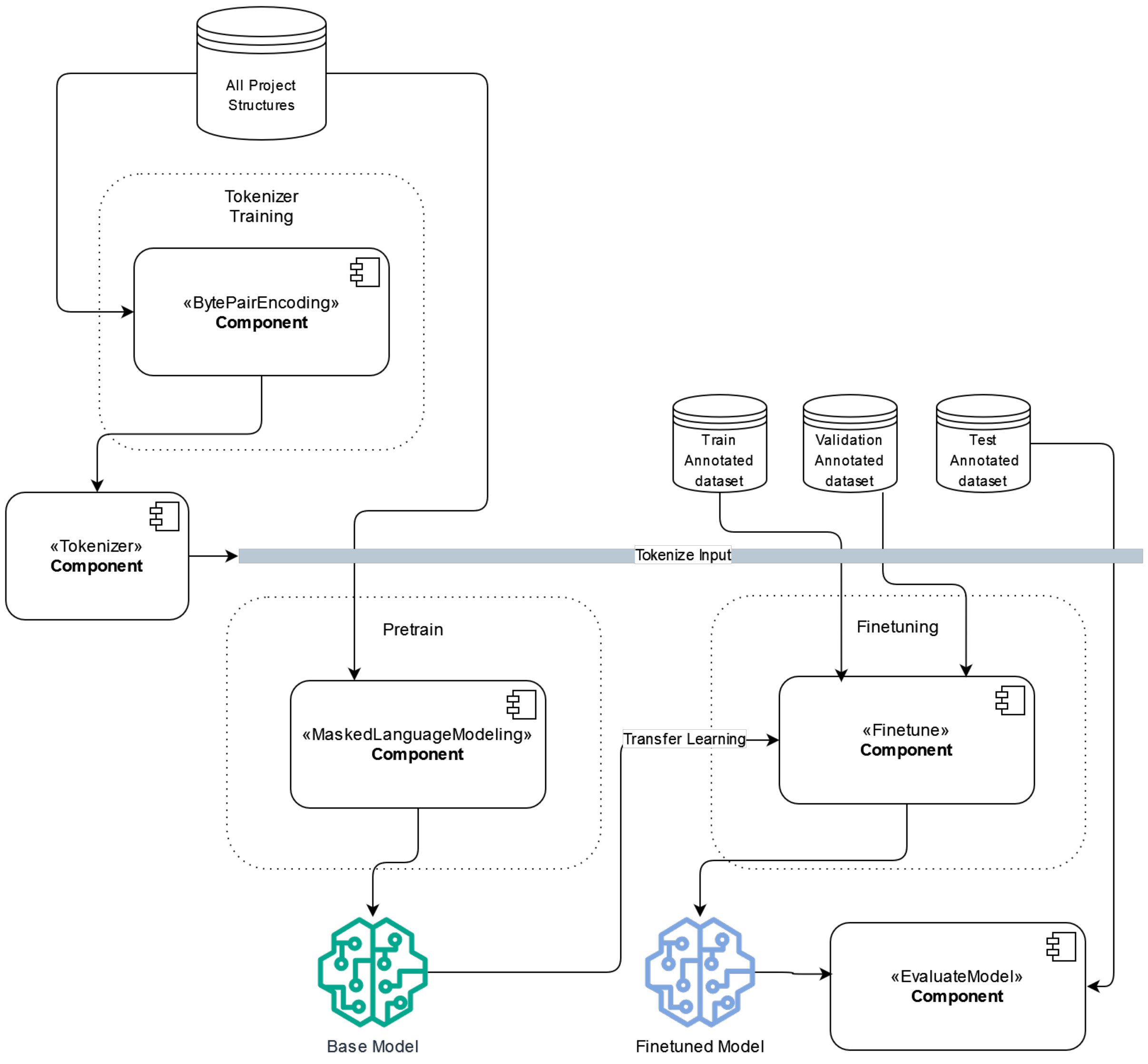


Figure 3.1: Model Training Process (High Level Overview)

3.2 Why Transformers?

Contextual Awareness: Transformers, through self-attention mechanisms, can weigh the importance of different parts of the input sequence, allowing them to understand the context and significance of each directory, file name, and their hierarchical position within a project structure.

Parallel Processing: Unlike recurrent neural networks, transformers process the entire sequence in parallel, significantly reducing training times and allowing for the handling of longer sequences more efficiently.

Scalability: The architecture scales well with the addition of more layers, enabling the model to capture more complex relationships in the data without a drastic increase in training time.

3.2.1 Quick Introduction to transformers

Transformers were introduced by Vaswani et al. in the paper ”Attention is All You Need” (2017) https://arxiv.org/abs/1706.03762 and have since revolutionized the field of NLP, as of today they are being explored in many more applications. The architecture is based on self-attention mechanisms, which allow the model to weigh the importance of different words in a sentence relative to each other, regardless of their distance (limited to the attention window). This capability enables transformers to capture long-range dependencies and contextual information more effectively than previous models like recurrent neural networks (RNNs) and convolutional neural networks (CNNs), refer to figure 3.2 to visualize the diagram.

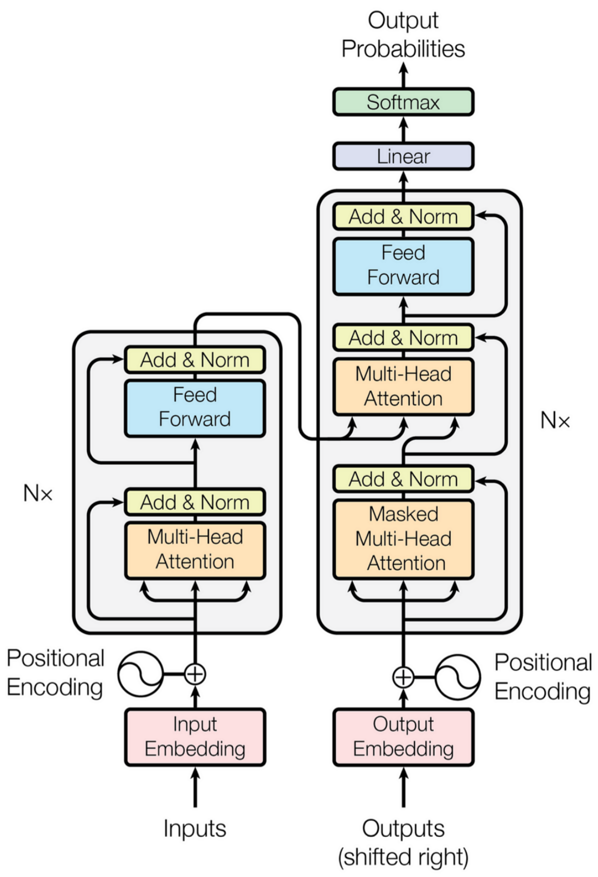


Figure 3.2: Transformer Architecture

A typical transformer model consists of an encoder and a decoder, each composed of multiple layers of self-attention and feed-forward neural networks. In many applications, such as text classification or translation, only the encoder or decoder might be used. The self-attention mechanism within these layers allows the model to process input sequences in parallel, significantly improving computational efficiency and scalability.

Encoder

Encoders are designed for tasks that involve understanding or interpreting input data to transform it into a more abstract representation. A classic example is text classification, where an encoder processes a sequence of words to understand its sentiment or topic without needing to generate any new text.

In tasks like sequence classification, when using transformers, only an encoder is needed. This is because the goal is to analyze the input and classify it without needing to generate new content. The transformer's encoder accomplishes this through its ability to weigh the importance of different parts of the input sequence. This feature allows it to understand contextual nuances, which is critical in accurately categorizing sequences.

Decoder

Decoders are designed for generating outputs based on some form of input representation. They excel in tasks where the model must produce a sequence or output that is a direct continuation or transformation of the input data. This functionality is essential in a variety of applications that require the generation of coherent and contextually relevant data from an 'encoded' state, note that an encoder is not required.

The GPT (Generative Pre-trained Transformer) models are classic examples of decoders only used in an autoregressive manner. Here, the model generates text by predicting the next word in a sequence based on the previous words. This process is not about understanding or classifying the input but about creating coherent and contextually appropriate text based on it.

Encoder-Decoder

Both components are utilized in tasks that involve both understanding the input and generating related outputs. This dual usage is typical in machine translation or speech recognition, where the input must first be comprehended and then transformed into a new format.

3.3 Considerations For AI Frameworks

TensorFlow and PyTorch are two of the most prominent libraries used in the field of artificial intelligence (AI), particularly for developing models like transformers. The role of an AI framework is to provide robust tools for numerical computation, data management, and model training. These frameworks support parallel computing, facilitating scalable and efficient training across multiple devices. Their extensive libraries and community-driven enhancements enhance the engineers productivity and innovation, making them essential for tackling complex AI challenges in various industrial and research settings.

TensorFlow

Developed by Google, TensorFlow, TensorFlow (2024); **?** is an open-source library designed for numerical computation using data flow graphs. It is well-suited for large-scale machine learning (ML) tasks and supports both CPUs and GPUs. TensorFlow is known for its powerful production-ready deployment options, which make it ideal for applications that require scalability and high performance. Its integration with the broader Google ecosystem, including tools like TPU support and TensorFlow Extended (TFX) for managing ML production pipelines, makes it a comprehensive choice for many industrial applications.

PyTorch

Originally developed by Facebook's AI Research lab, PyTorch, PyTorch (2024), has gained immense popularity due to its user-friendly interface and dynamic computational graph, which allows for more intuitive coding of complex AI models like transformers. PyTorch's design is highly favored for academic research and development because it facilitates easy and fast prototyping. Its dynamic nature (where computations can be changed on the fly) is particularly useful for projects where model architecture experiments are frequent.

Why PyTorch is Highly Used in the Industry

While TensorFlow was the frontrunner in adoption, PyTorch has seen increasing use in the industry for several reasons:

- **Ease of Use:** PyTorch's more Pythonic interface and support for dynamic computation graphs make it highly intuitive for developers and researchers.
- **Flexibility:** The ability to adjust and view every part of the model as it runs makes PyTorch versatile and adaptable for research and development.
- **Strong Community and Support:** PyTorch benefits from stron. community engagement, comprehensive tutorials, and support from industry leaders like Meta (Author of the best open source autoregressive model 'Llama3'), the company behind Facebook, which contribute to its rapid adoption.
- **Research to Production:** With the introduction of PyTorch Lightning and TorchServe, the transition from research to production has been streamlined, enhancing its attractiveness for industrial applications.

Overall, both TensorFlow and PyTorch offer robust functionalities for the development of AI technologies, particularly in handling complex tasks like training transformers. For this internship, only Pytorch was used.

3.4 An Introduction to Huggingface

Huggingface, Hugging Face (2024), is a company and community in the field of artificial intelligence. Huggingface has contributed significantly to the popularity and accessibility of transformer models through its library called transformers. This open-source library provides pre-trained models that can be fine-tuned on specific tasks, allowing developers and researchers to implement state-of-the-art NLP models with minimal effort. Some popular models available through Hugging Face include BERT, GPT (and its successors), and T5.

Tokenization is a fundamental step in processing text data for machine learning models. Hugging Face also offers a tokenizers library specifically designed for this purpose. This library is highly optimized for performance and can handle various tokenization methods used by modern NLP models, such as Byte-Pair Encoding (BPE), WordPiece, and SentencePiece. The tokenizers library is crucial for preparing text data to be compatible with transformer models.

One of the standout features of Hugging Face is its vibrant community. The platform encourages sharing and collaboration, enabling users to contribute their own trained models and to utilize models shared by others. This community-driven approach helps in democratizing AI technologies, making cutting-edge models available to a broader audience. Users range from academic researchers to industry professionals, all contributing to the enhancement and application of transformer technology.

3.5 AI System Services

This section outlines the main services of the designed AI system.

1. Tokenizer Training
2. Pretrain Model Training
3. Finetune Model Training

3.5.1 Tokenizer

In the context of data processing and machine learning, tokenization involves segmenting data into smaller, manageable units called tokens. These tokens are essential for the efficient and effective processing of information by algorithms, especially in transformers.

Transformers, a type of neural network architecture, excel in handling sequential data by transforming it into numerical representations (tokens) with the help of a tokenizer.

In the context of natural language processing (NLP), a token is somewhere between words and characters, depending on vocabulary size and tokenization rules, a token can be multiple words.

The tokenizer starts to abstract the context, **it's all about tokens and how they relate to each other**, making the transformer capable of processing everything that can be mapped into tokens.

In theory, a token can be a byte (0-255) allowing the transformer to process any type of content, but this would require a really high context length and attention window, exponentially increasing the model's computational requirements. Most state-of-the-art models are currently being trained with vocabulary sizes of (32k-128k).

A lot of research is being pursued in the field of tokenization. A few notable papers are mentioned below:

- Xue et al. (2022), outlines the benefits and implementation details of byte-level processing in transformer models.
- Yang (2024), explores the evolution and efficiency of various tokenization methods.

3.5.2 Base Model

For the specified task of analyzing project structures, a transformer model can derive substantial advantages from pre-training specifically on project-related data. Given that the model in question is not a large language model and is exclusively utilized for project structure analysis, this targeted pre-training approach ensures that the model is finely tuned to recognize and process the unique patterns, terminology, and structural characteristics inherent in project data.

The model was pre-trained through a technique called MLM (masked language modeling), a type of self-supervised learning. This step helps the model learn a rich representation of structural patterns, relationships, and possibly the implied functionality of different project components.

Why is it required?

The first reason seems obvious: there are no base models optimized for this task and large language models would have a really small percentage of unique tokens and/or relations that are relevant for the task.

A model that is already pre-trained with a rich representation of the data was not available in open source communities.

Now one might ask, why not start with a process that does it all, pre-training and fine-tuning at the same time? There are many reasons why this is discouraged. A few are mentioned:

- Large datasets are required
- Longer training time
- Increased computation resources
- The risk of overfitting is too high
- Loss of transfer learning benefits
- Lower starting performance
- Difficulty in learning complex patterns

3.5.3 Finetuned Model

Finetuning a transformer model for this specific task of multi-label sequence classification involves adapting the previous pre-trained transformer to classify sequences (texts).

In this stage the main objective is to make the model capable of classifying project types using the previous knowledge gathered from the pre-training stage.

This is done by adding a classification layer (Fully Connected), also refered to as Linear layer or Dense layer, to the model and training it on labeled data, allowing the model to learn to predict multiple labels for each input sequence, refer to figure 3.3 to visualize a neural network example with a liner layer.

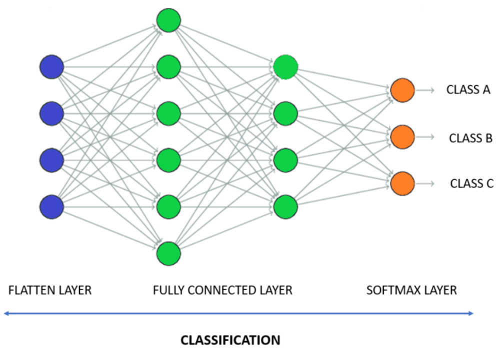


Figure 3.3: Example Simple Fully connected layer

DataSets

Three DataSets were designed to train the model.

- Training Dataset: Used to train the machine learning model.
- Validation Dataset: This is used to evaluate the performance of the model at epoch/-batch level.
- Test Dataset: This is used to evaluate the performance of the model after it has been trained. It should never be used during the training process to ensure an unbiased evaluation of the model's performance.

DataSet Structure

- Text
- Labels [0 0 1 1] (Vector with labels)

Label 0 $\rightarrow$ Backend
Label 1 $\rightarrow$ Frontend
Label 2 $\rightarrow$ Mobile
Label 3 $\rightarrow$ Use of Rest

3.5.4 Inference API

The Inference API, designed using REST principles, offers a versatile platform for machine learning inference. The API will serve as the principal means of interacting with the system, providing users with multiple options for inference (with a git repository or client built tree), including tokenization process visualization. The client should not download the AI model wich has a size of 500MB.

3.6 Model Architecture

The model architecture described here pertains to the Longformer, (Beltagy et al., 2020), a variant of the Transformer designed for processing long sequences by employing an efficient attention mechanism. The Longformer utilizes a combination of global attention on selected tokens and sliding window attention, enabling it to handle sequences much longer than those manageable by standard Transformer models. This architecture is particularly beneficial for tasks requiring the handling of large texts, such as document classification, summarization, and question answering.

For the problem at hand "project classification" where project structures are analyzed, large projects generate a significant amount of tokens, and a region of that project could be misclassified due to small context size and or sequence length, the use of a longformer addresses the context lenght limitation, refer to figure 3.4 to visualize common attention mechanisms.

The **sliding window attention** is a form of localized attention where each token only attends to a fixed number of adjacent tokens. This creates a window around each token, limiting the scope of attention to just the nearby context. The window size is a configurable hyper-parameter of the model. This mechanism drastically reduces the **complexity** from $O(n^2)$ to $O(n \cdot w)$ where w is the fixed window size.

The Longformer introduces **gobal attention** on selected tokens. This means that some tokens (chosen based on their importance to the task, `[CLS]`) can attend to all other tokens in the sequence and all other tokens can also attend to them. This helps the model maintain an awareness of crucial parts of the input regardless of their position, somewhat mitigating the locality limitation imposed by the sliding window attention.

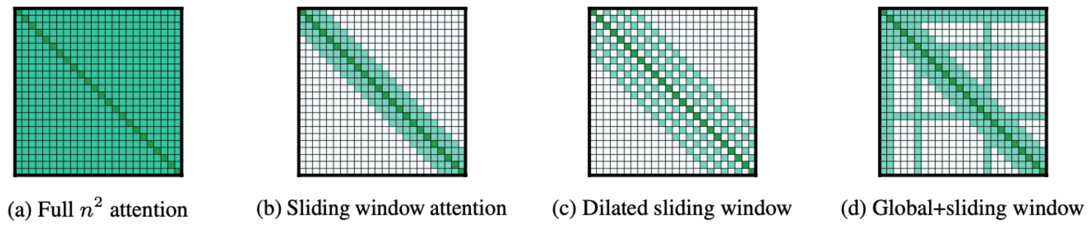

(a) Full n^2 attention
(b) Sliding window attention
(c) Dilated sliding window
(d) Global+sliding window

Figure 3.4: LongFormer Attention mechanism (d)

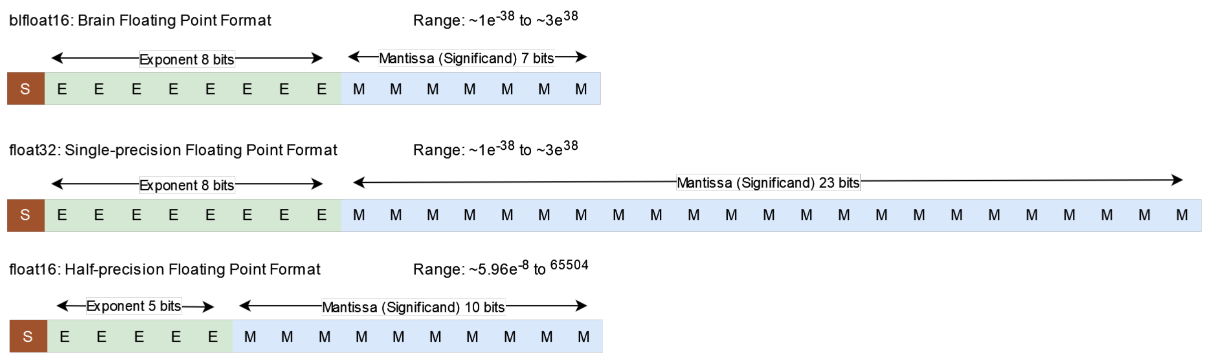

blfloat16: Brain Floating Point Format
Range: ~1e-38 to ~3e38
Exponent 8 bits
Mantissa (Significand) 7 bits
S E E E E E E E E M M M M M M M
float32: Single-precision Floating Point Format
Range: ~1e-38 to ~3e38
Exponent 8 bits
Mantissa (Significand) 23 bits
S E E E E E E E E M M M M M M M M M M M M M M M M M M M M M M M
float16: Half-precision Floating Point Format
Range: ~5.96e-8 to 65504
Exponent 5 bits
Mantissa (Significand) 10 bits
S E E E E E M M M M M M M M M M

Figure 3.5: Floating Point Data Types.

3.7 Data Types

In the field of machine learning, the choice of data types plays a crucial role in designing and optimizing algorithms. This section of the report focuses on different floating-point data types commonly used in machine learning computations, namely bfloat16, float32, and float16, as well as the concept of mixed precision. These data types are integral in managing the trade-offs between memory usage, performance, and the accuracy of the results, refer to figure **??**

3.7.1 float32

Float32, also known as single-precision floating-point, has been the standard data type in traditional computing due to its balance between range and precision. It consists of 1 sign bit, 8 exponent bits, and 23 mantissa bits. In machine learning, float32 is often used when high numeric precision is necessary, such as in tasks involving detailed image processing or complex calculations. However, its use demands more memory and computational resources compared to bfloat16 or float16.

3.7.2 float16

Float16 or half-precision floating point, includes 1 sign bit, 5 exponent bits, and 10 mantissa bits. This data type uses significantly less memory and allows for faster processing speeds, making it suitable for scenarios where high throughput is more critical than high precision. However, its reduced numerical precision and smaller exponent range make it prone to issues like overflow and underflow during calculations, which can affect the model's performance and accuracy.

3.7.3 bfloat16

The bfloat16 (Brain Floating Point) data type is a 16-bit floating point format that is particularly tailored for machine learning applications. It comprises 1 sign bit, 8 exponent bits, and 7 mantissa bits. This configuration offers a wide dynamic range as extensive as that of the 32-bit float (float32), making it particularly advantageous for machine learning models where numerical range is more critical than precision. The use of bfloat16 can lead to significant reductions in memory consumption and can speed up the training and inference phases without a substantial sacrifice in the model's accuracy.

3.7.4 Mixed Precision

Mixed precision training utilizes both 16-bit and 32-bit floating-point types to optimize machine learning workflows. By using 16-bit floating-point types (such as `bfloat16`) for parts of the computation where high precision is not crucial, we can increase computational speed and reduce memory usage. Conversely, 32-bit floating-point types (`float32`) are used in areas requiring higher numerical precision to maintain accuracy. This approach balances performance and accuracy, enhancing computational efficiency and enabling the use of larger models or datasets that would not fit in memory with only higher precision types.

The use of mixed precision is particularly advantageous because it allows for:

- **Increased training speed**: By reducing the data precision requirements, mixed precision allows for faster processing and less strain on memory resources.
- **Reduced memory usage**: Lower precision types use less memory, enabling larger batches or models to be loaded into the same physical memory space.
- **Maintained accuracy**: By carefully allocating precision where needed, the model's accuracy can be preserved even with reduced data precision.

However, there are also disadvantages:

- **Potential for numerical instability**: Lower precision can lead to issues like underflow or overflow during computations, potentially impacting training stability and model convergence.
- **Compatibility issues**: Some algorithms may not benefit from or be compatible with mixed precision training.
- **Complexity in implementation**: Managing different precisions within the same model can increase the complexity of the code and the debugging process.

How To Use

The first code sample demonstrates how to enable mixed precision in a training loop using NVIDIA's autocast feature with PyTorch:

```
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), ...)

for input, target in data:
    optimizer.zero_grad()

    # Enables autocasting for the forward pass (model + loss)
    with torch.autocast(device_type="cuda"):
        output = model(input)
        loss = loss_fn(output, target)

    # Exits the context manager before backward()
    loss.backward()
    optimizer.step()
```

Listing 3.1: Autocast to enable mixed precision.

In this code, `torch.autocast` is used to automatically manage the precision of certain operations, optimizing them for performance. The forward pass of the model, including the calculation of loss, is done under the autocast context, which may use `bfloat16`. Once the context is exited, the backward pass uses the full precision for gradient calculations to ensure accuracy.

The second code sample integrates `GradScaler` to handle gradients in mixed precision scenarios:

```
scaler = GradScaler()

with autocast():
    output = model(input)
    loss = loss_fn(output, target)

scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
```

Listing 3.2: Using GradScaler to manage gradients in mixed precision.

Here, `GradScaler` is utilized to adjust the gradient scale automatically, preventing issues like underflow during the backward pass. By scaling up the gradients before backpropagation and scaling them down afterwards, the model can effectively learn even with reduced precision in certain computations.

These examples show how mixed precision can be practically implemented to enhance the efficiency of training deep learning models without sacrificing performance quality.

4. Implementation Details

This chapter outlines the implementation details of the project undertaken during the internship. It describes the deployment of various services essential for delivering a comprehensive solution that encompasses AI services, a backend web server, and a frontend website.

4.1 Services Overview

This section offers a concise overview of the primary services constituting the system, designed in alignment with a microservices architecture.

- Pretrain - This service is responsible for the initial training of a model from scratch, which includes training a tokenizer alongside a masked language model (MLM).
- Finetune - Designed to finetune a pre-existing base model tailored for classification tasks. This service dynamically adjusts the model by incorporating annotated data fetched from the web server.
- Web Server - This component features a REST API that facilitates a range of functions including project insertion, annotation management, tokenization visualization, inference requests, and feedback processing. It supports admin authentication and incorporates local large language model (LLM) capabilities via websockets.
  - Postgres - Utilized as the main Database Management System (DBMS) for persistent data storage.
  - Redis - Employed as the primary in-memory database to manage AI datasets and house a blacklist of revoked JSON Web Tokens (JWT).
- Website - This interface serves as the front-end component, providing user interaction and visualization tools.
- Utility Scripts - A series of scripts developed to automate various tasks including data annotation through LLM, database backups, and integration with the GitHub API.
- Local LLM - A standalone service offering an API compatible with OpenAI, allowing the use of a local LLM independently from the main web server.
- Documentation - Comprehensive documentation was prepared to support the system's deployment and maintenance, including model AI model trainig, ensuring clarity and ease of use for system administrators and developers.

4.2 Pretrain Service

This section details the engineering process of the pretrain service.

1. **Tokenizer Engineering**
    - *Tokenizer Training*: Develop a tokenizer capable of breaking text into manageable units (tokens) that the model can understand.
    - *Tokenizer Component*: Implement the tokenizer as a reusable component that converts raw text into a sequence of tokens.
2. **Pretrain**
    - *Input Tokenization*: Use the tokenizer to convert input data into token sequences.
    - *Masked Language Modeling*: Train the model to predict missing tokens in a sequence to learn contextual relationships between tokens.
    - *Training Epochs*: Run multiple cycles of training to allow the model to iteratively learn from a vast amount of text data.
    - *Base Model*: Establish a base version of the model, which understands general language but is not yet specialized for any specific task.

4.2.1 Tokenizer Engineering

Tokenization, as discussed in Chapter 3 of ”System Modeling,” is a fundamental process for transformers. The most prevalent method for training a tokenizer is through Byte Pair Encoding (BPE). This technique plays a critical role in efficiently preparing text data as input to the model.

Byte Pair Encoding (BPE) is a data compression technique originally designed for file compression but has been effectively adapted in the field of natural language processing (NLP), particularly in tokenizer training.

Byte Pair Encoding operates on a basic principle: it iteratively merges the most frequent pair of bytes (or characters) in a given text into a single, new byte (or character) that did not exist in the text before. This process is repeated until a set number of merges have been completed or a specified vocabulary size is achieved.

BPE Training Overview

1. **Initialization**: BPE starts with a vocabulary consisting of the character set of the text, where each character is treated as a word or token.
2. **Pair Identification**: The algorithm scans the text to find the most frequently occurring adjacent pairs of characters (tokens).
3. **Token Merging**: Each occurrence of the most common pair is replaced with a new token that uniquely represents this pair. This new token is added to the vocabulary.

4. **Iteration**: Steps 2 and 3 are repeated for a specified number of iterations or until the vocabulary reaches a predetermined size.

Consider the string “aaabdaaabac”. The initial vocabulary is {a, b, d, c}. Suppose the first pair identified is (’a’, ’a’), which is the most frequent. After the first iteration, ’aa’ becomes a single token ’A’, and the string transforms to “AAbdAAAbAc”. The vocabulary updates to {A, b, d, c, a}.

Byte Pair Encoding represents an advancement in tokenizer training, striking a balance between vocabulary size and the granularity of text representation. Its application extends across multiple domains within NLP, proving essential for contemporary language-based models. By optimizing how text is segmented into manageable pieces, BPE not only improves the operational efficiency of models but also enhances their ability to understand and generate any type of language. To visualize the process refer to figure 4.1.

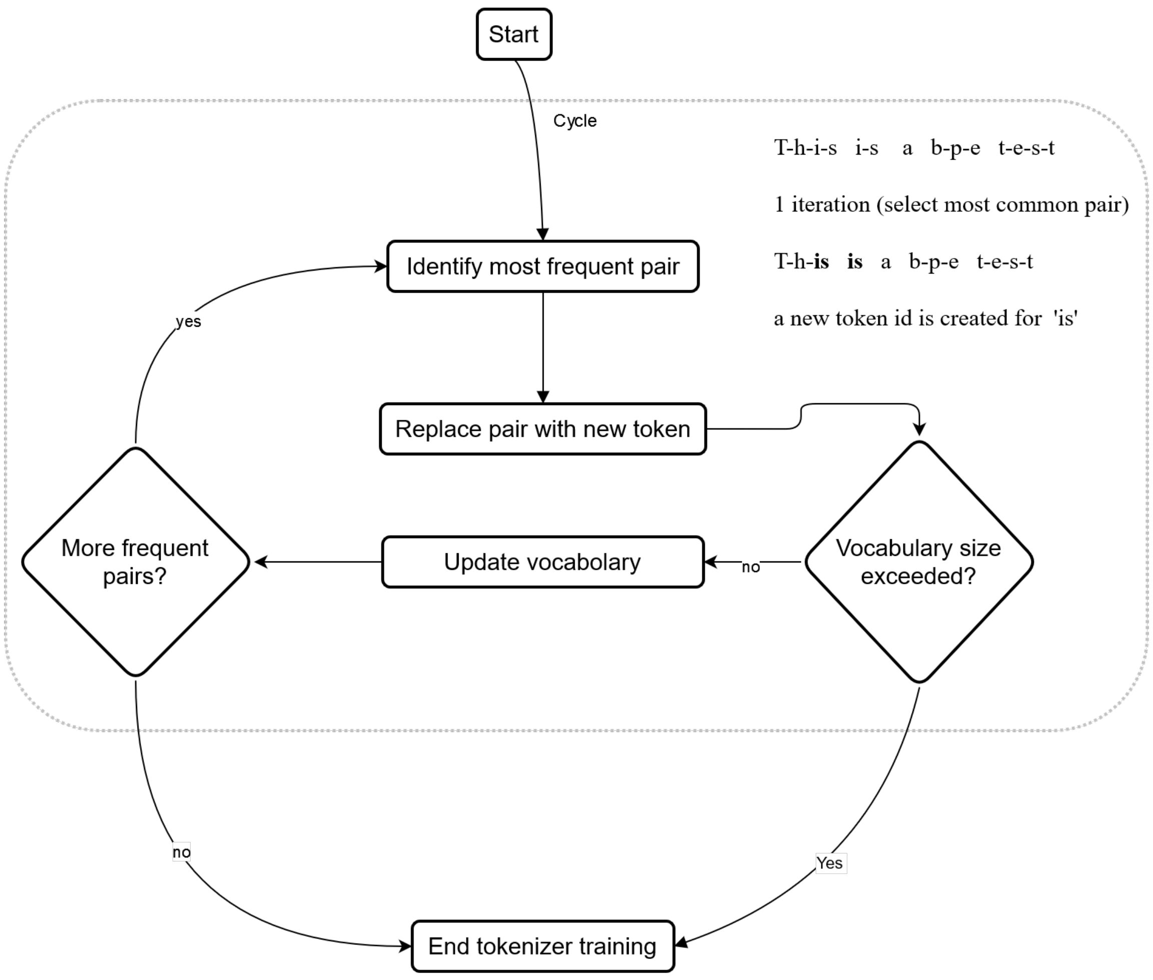

Start
Cycle
T-h-i-s i-s a b-p-e t-e-s-t
1 iteration (select most common pair)
T-h-**is** **is** a b-p-e t-e-s-t
a new token id is created for 'is'
Identify most frequent pair
yes
Replace pair with new token
More frequent pairs?
Update vocabolary
no
Vocabulary size exceeded?
no
Yes
End tokenizer training

Figure 4.1: Byte Pair Encoding (BPE) Algorithm Diagram

Special Tokens

Special tokens play a fundamental role in this process by helping models understand and manage various aspects of the text. These tokens include [CLS] and [SEP], which respectively mark the beginning of a sequence and separate segments within the text. Additionally, [PAD] tokens are used to ensure uniform sequence lengths in batch processing, and [MASK] tokens facilitate training in models like BERT by hiding specific words to predict them based on context.

Overall, special tokens are crucial for the model's ability to interpret and process language effectively. Table 4.1 is present with all the special tokens used in the model.

Special Token	**Description**
[CLS]	A classification token added at the beginning of the input sequence, whose final hidden state is often used for classification tasks.
[SEP]	A separator token used to distinguish different segments within the input sequence.
[PAD]	A padding token used to ensure all input sequences are of the same length, facilitating batch processing.
[MASK]	A token used in masked language modeling tasks, where certain tokens in the input are replaced with [MASK] and the model is trained to predict the original tokens.
[UNK]	The ”unknown” token is used whenever an unknown or out-of-vocabulary word is encountered during processing.
[BOS]	The ”beginning of sequence” token is used to mark the start of a sequence. It is crucial in models that need explicit signals to denote sequence start, particularly in generative tasks.
[EOS]	The ”end of sequence” token is similarly used to indicate the termination of a sequence. This helps the model in recognizing when to stop processing or generating tokens in tasks like text generation.

Table 4.1: Special Tokens

The example script creates and trains a tokenizer using Byte Pair Encoding (BPE) in the most simple way I can think of. It initializes the tokenizer with parameters such as vocabulary size and special tokens like `[unk]` for unknown words and `[s]` for the start of a sentence. The tokenizer is trained on a given list of text files and then saved to a file named `tokenizer.json`. Finally, it returns a `PreTrainedTokenizerFast` object, which is optimized for tokenization tasks in various machine learning applications and most importantly easy to integrate with Hugging Face's transformers library.

```
def build_tokenizer(
    train_files: list[str],
    vocab_size: int = 2**15,
    unk_token: str = "[unk]",
    bos_token: str = "[s]",
    eos_token: str = "[/s]",
    pad_token: str = "[pad]",
    mask_token: str = "[mask]",
    cls_token: str = "[s]",
    sep_token: str = "[/s]",
    add_tokens: List[str] = [],
) -> PreTrainedTokenizerFast:

    special_tokens = [
        unk_token,
        mask_token,
        # (...)
    ] + add_tokens

    tokenizer = Tokenizer(BPE(unk_token=unk_token))
    tokenizer.pre_tokenizer = Whitespace()

    trainer = BpeTrainer(special_tokens=special_tokens, vocab_size=vocab_size)
    tokenizer.train(train_files, trainer)

    tokenizer.save("tokenizer.json")

    return PreTrainedTokenizerFast(
        tokenizer_object=tokenizer,
        unk_token=unk_token,
        bos_token=bos_token,
        # (...)
    )
```

Listing 4.1: Example Tokenizer Component Training

4.2.2 Model Hyperparameters

The choice and tuning of hyperparameters in Longformer are critical for optimizing its performance, scalability, and applicability to the task.

Hyperparameters are external configurations of a model that govern its architecture and training process. They are not learned from the data but set before the training begins. For Longformer, hyperparameters include the number of attention heads, the attention window size, model dimensions, vocabulary size (requires tokenization change), among others. Model Hyper-parameter tuning can have sereval effects, mainly:

- Performance
- Effectiveness
- Generalization Capabilities
- Task-Specific Adaptations
- Sequence Processing Size
- Attention Complexity
- Chances of Overfitting
- Numerical Stability

Table 4.2 outlines the principal model hyper-parameters used for the longformer configuration.

Hyperparameter	**Description**
vocab_size	Specifies the number of unique tokens the model can recognize, determined by the tokenizer used on the base model. If training a custom base, this should be the number of unique tokens during the BPE tokenizer build plus the total number of special tokens.
attention_window	This parameter is an array defining the local attention window size for each transformer layer, with typical setups utilizing powers of 2 (e.g., 64, 128, 256), which help strike a balance between computational demands and performance efficacy.
hidden_size	Indicates the dimensionality of the hidden layers.
num_hidden_layers	Total count of hidden layers within the transformer architecture.
num_attention_heads	The number of attention heads in each transformer layer, facilitating multiple perspectives of attention in each layer.
intermediate_size	Dimension of the intermediary (feedforward) layer in the transformer.
hidden_act	Specifies the activation function, generally “gelu”.
hidden_dropout_prob, attention_probs_dropout_prob	Dropout probabilities for hidden layers and attention mechanisms to prevent overfitting.
max_position_embeddings	Defines the maximum sequence length the model can process.
initializer_range	Standard deviation of the truncated_normal_initializer for initializing all weight matrices.
layer_norm_eps	A small constant added to the denominator for numerical stability in layer normalization.

Table 4.2: Base Model Hyperparameters

4.2.3 DataSet and Data Loading

To correctly prepare for pretraining the Longformer model, all training samples should be loaded effectively. This can be accomplished by using a `DataSet` from `torch.utils.data`, which enables fast and optimized data loading. By integrating a tokenizer and applying rules to the data, the preparation process is further enhanced, ensuring the model is trained efficiently and effectively. To better understand the process inspect the code sample 4.2.

When the tokenizer processes a piece of text, it converts it into token IDs based on a pre-defined vocabulary. The parameter `max_length` enforces a fixed size for all sequences, truncating longer texts and padding shorter ones to this length. The `.squeeze()` method is used to remove any extra dimensions from the tensor, making it suitable for processing in the model.

```python
class PreTrainDataset(Dataset):
    def __init__(
        self,
        texts: List[str],
        tokenizer: PreTrainedTokenizerFast,
        max_length: int
    ):
        self.tokenizer = tokenizer
        self.texts = texts
        self.max_length = max_length

    def __len__(self) -> int:
        return len(self.texts)

    def __getitem__(self, idx: int) -> dict:
        encoded = self.tokenizer(
            self.texts[idx],
            max_length=self.max_length,
            truncation=True,
            padding="max_length",
            return_tensors="pt",
        )
        return {
            "input_ids": encoded["input_ids"].squeeze(),
            "attention_mask": encoded["attention_mask"].squeeze(),
        }
```

Listing 4.2: Example Pretrain DataSet initialization linked to a Tokenizer

The *attention_mask* is a binary tensor indicating which tokens in the *input_ids* are meaningful data and which ones are padding added to achieve uniform sequence lengths. A value of 1 indicates that the corresponding token is part of the actual data, and a 0 indicates a padding token.

This mask ensures that the model does not process padding tokens as part of the input sequence during training or inference. Like the *input_ids*, it is also adjusted by `.squeeze()` to match the dimensionality expected by the model.

These two elements are fundamental for models that use attention mechanisms, like those based on the Transformer architecture (e.g., BERT). The attention mechanism in these models uses the *attention_mask* to weigh the influence of each token differently. Tokens represented by a 1 in the mask influence the model's output, while those represented by a 0 do not affect the computation beyond serving as positional placeholders.

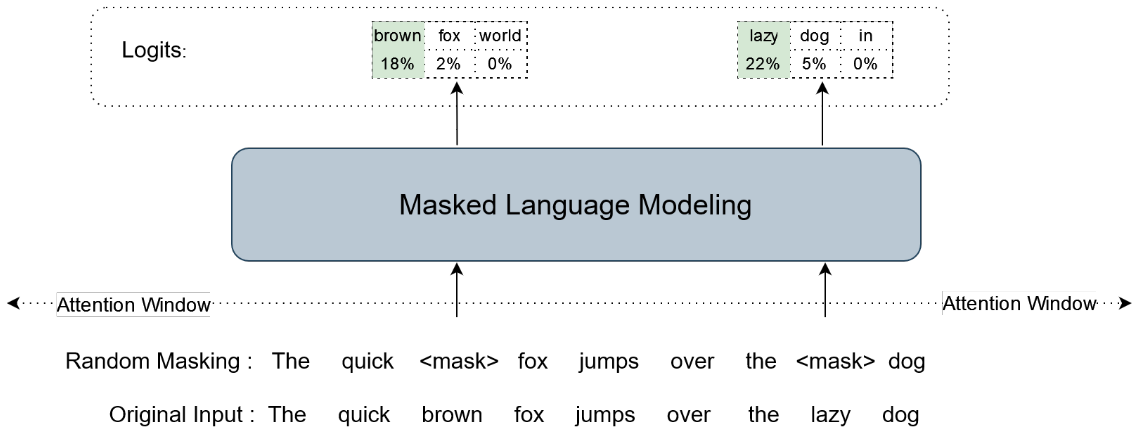

Logits:
brown 18%
fox 2%
world 0%
lazy 22%
dog 5%
in 0%
Masked Language Modeling
Attention Window
Attention Window
Random Masking : The quick <mask> fox jumps over the <mask> dog
Original Input : The quick brown fox jumps over the lazy dog

Figure 4.2: Masked Language Modeling (MLM) Diagram.

4.2.4 Masked Language Modeling

Masked Language Modeling (MLM) is a technique used to train the encoder part of transformers. MLM is the foundational algorithm used to train the base model of this project, it involves artificially masking part of the input data during training. The objective is for the model to predict the missing tokens based on the context provided by the other tokens in the sequence. This approach is fundamental in training language models to understand the sequences, refer to figure 4.2 to visualize the concept.

Concept and Implementation

MLM is a form of self-supervised learning where no external labels are required. Instead, the model generates its own labels from the input data. The process involves the following steps:

1. Tokenization: The input text is split into tokens, which can be words, parts of words, or special characters.
2. Masking: A certain percentage of these tokens (In this case 32%-8%) are randomly selected and replaced with a special [MASK] token. The selection of which tokens to mask is random, but it usually avoids masking special tokens that provide structural information about the sequence. The masking probabily changes at epoch level to start with basic understanding (structural patterns) to patters that are complex or less common. Dynamic masking is specific to this project's implementation and not a rule in MLM, the masking probability is typically defined at a static 15% percentage, this change improved training speed.
3. Prediction: The model then attempts to predict the original value of the masked tokens based solely on the context provided by the non-masked tokens in the sequence.
4. Training: The model's predictions are compared to the actual original tokens. The difference between the prediction and the actual token (the error) is used to adjust the model's parameters through backpropagation.

MLM has been popularized by models like BERT (Bidirectional Encoder Representations from Transformers) and its variants, which use MLM as a core part of their training process. By predicting masked words, the models learns a deep, contextual representation of language that captures nuances such as syntax and semantics more effectively than models trained on traditional left-to-right or right-to-left predictions alone.

Advantages

- Contextual Understanding: MLM allows the model to use both left and right context, which helps in understanding the full meaning of sentences more accurately.
- Flexibility: It can be used with any language without modification to the underlying model architecture, making it highly adaptable.
- Efficiency: It focuses the model's learning efforts on a subset of tokens at each training step, which leads to more efficient learning patterns.

Challenges

- Computational Cost: Predicting all tokens simultaneously requires significant computational resources, especially with larger datasets and complex model architectures.
- Data Requirement: Effective MLM training requires large and diverse datasets to ensure that the model can generalize well from its training context to new, unseen contexts.

Masked Language Modeling represents a significant advancement in how machines understand the input sequence. By predicting tokens hidden from a given text snippet, MLM trains models to deduce word meanings and relationships from surrounding text, leading to more robust and versatile language models. This technology continues to be at the heart of many state-of-the-art NLP systems, contributing to advancements in machine translation, text summarization, and beyond.

During the internship, significant improvements were observed in the base model's training metrics. The Masked Language Model (MLM) masking probability decreased from 32% to 8% to better understand complex relations at the last epochs, and start with basic understanding of relations (common patterns) at the initial epochs. The loss reduced from 12 to 0.7, training was not extended beyond this point to prevent overfitting, which is particularly detrimental at the pretraining stage given the dataset size of 60,000 project structures (samples).

4.2.5 Causal Language Modeling

Causal Language Modeling (CLM), also known as autoregressive language modeling, is a technique used in natural language processing where the model learns to predict the next token in a sequence given the tokens that preceded it. This approach is fundamental for models that generate text, as it enables them to produce coherent and contextually

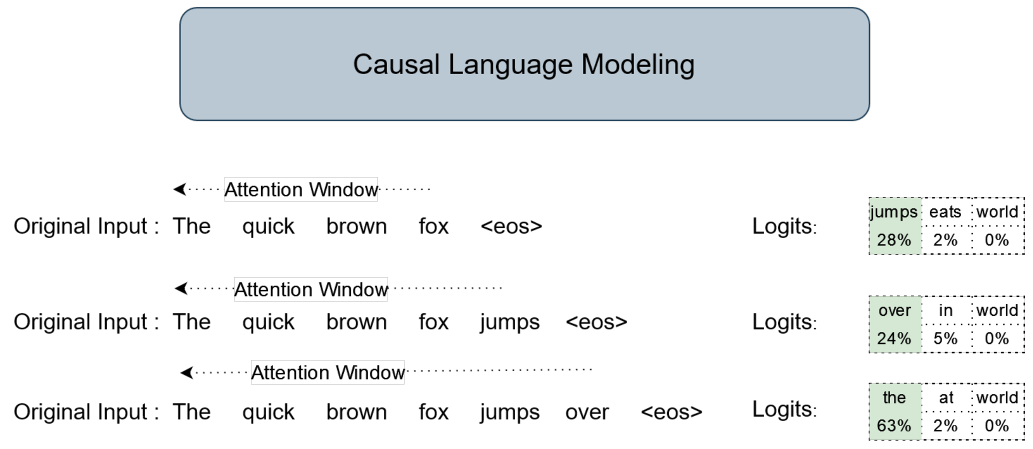


Figure 4.3: Causal Language Modeling (CLM) Diagram.

appropriate continuations of the input text. This was not used in this project, but it is a really important concept to understand how models are trained. A change on the training algorithm can adpat this project to autoregressive taks with minimal effort, refer to figure 4.3 to visualize the concept.

Concept

CLM operates under a straightforward principle: predict the future based on the past. The process involves the following steps:

1. Tokenization: The input text is split into tokens, similar to the approach in masked language modeling.
2. Prediction Task: The model is trained to predict each token sequentially, based on the tokens that have come before it in the sequence. For instance, given the sequence ”The quick brown fox”, the model might predict ”jumps” as the next token.
3. Training: The model's predictions are made one token at a time, and it is trained to minimize the error between its predictions and the actual tokens that follow.

CLM is a core technique for generating text and is used by models like GPT (Generative Pre-trained Transformer). The autoregressive nature of these models allows them to generate lengthy and coherent text passages, as each token generated can influence the prediction of the next token, maintaining context and logical flow throughout. A comprehensive comparison between MLM and CLM was performed.

Direction of Context

- MLM: Uses bidirectional context, meaning the model learns from both the preceding and following tokens around a mask token. This provides a richer understanding of the language context.
- CLM: Uses unidirectional context, where each token prediction can only utilize the previous tokens in the sequence. This models the way humans often generate language but can limit the understanding of the context compared to MLM.

Training Objective

- MLM: The objective is to fill in the blanks (masked tokens) using the surrounding words as context. The model's focus is on understanding and predicting based on a given context rather than generating new text.

- CLM: The goal is to generate the next token in the sequence, focusing on production of text. This sequential prediction trains the model to generate coherent and contextually appropriate text.

Use Cases

- MLM: Particularly useful in tasks that require a deep understanding of language structure and context, such as question answering, named entity recognition, and other forms of language understanding tasks.

- CLM: Ideal for applications that require token ('text') generation, such as creating content, composing poetry or more generally, bytes, and other creative writing tasks.

Performance and Efficiency

- MLM: Often requires more complex model architectures and greater computational resources due to its bidirectional nature. However, it can lead to a deeper understanding of language nuances.

- CLM: Can be more straightforward to implement and computationally efficient in scenarios where only forward context is necessary. However, it may miss some nuances of language that come from a broader context.

MLM excels in understanding and contextual analysis, making it suitable for comprehensive language tasks, while CLM excels in generating coherent sequences of text, suitable for any task where creative or predictive text generation is needed. Choosing between these models typically depends on the specific requirements of the application, whether it leans more towards understanding language or generating it.

The Understanding these two concepts and the component to which they relate, encoder or decoder, is foundational when working with transformers, particularly, when pretaining a base model.

4.2.6 Loss Calculation and Optimizer

This section briefly outlines the rationale for choosing the loss function and optimizer in training for MLM. Cross-Entropy Loss was selected as the loss function, and AdamW was chosen as the optimizer. The reasons for these choices are provided below.

Cross Entropy Loss

Cross-entropy loss is an effective loss function for Masked Language Model (MLM) training. In MLM cross-entropy loss measures the performance by comparing the predicted probabilities to the actual words.

The loss function increases when the predicted probability of the correct token is low, and decreases when the predicted probability is high. In other words, if the model accurately predicts the masked token, the loss is low, if it predicts incorrectly, the loss is high.

Cross-entropy loss is particularly suitable for MLM due to its simplicity in handling the probabilistic nature of token predictions.

Adaptive Moment Estimation with Weight Decay (AdamW)

AdamW, Loshchilov & Hutter (2017), is an optimization algorithm that extends the Adam optimizer by decoupling weight decay from the gradient updates. It was proposed by Ilya Loshchilov and Frank Hutter in their paper Decoupled Weight Decay Regularization (2017).

Key Concepts

- **Adam Optimizer**:
  - Combines the advantages of two other extensions of stochastic gradient descent: AdaGrad and RMSProp.
  - Maintains per-parameter learning rates that are adapted based on the first and second moments of the gradients.
  - Efficient for large-scale data and high-dimensional parameter spaces.
- **Weight Decay**:
  - A regularization technique used to prevent overfitting by adding a penalty to the loss function proportional to the magnitude of the weights.
  - In the traditional Adam implementation, weight decay is incorporated directly into the gradient updates, which can lead to suboptimal regularization.
- **Decoupled Weight Decay (AdamW)**:
  - Separates the weight decay from the gradient updates.
  - This decoupling allows for better control over regularization and often leads to improved performance.
  - The weight update rule in AdamW is given by:

    $$\theta_t \leftarrow \theta_t - \alpha \left(\frac{m_t}{\sqrt{v_t} + \epsilon} + \lambda \theta_t \right)$$

    where θ_t are the model parameters, α is the learning rate, m_t and v_t are the first and second moment estimates, ϵ is a small constant for numerical stability, and λ is the weight decay coefficient.

Benefits of AdamW

- Improved Regularization
- Adaptive Learning Rate
- Stable Training
- Numerical Stability
- Ease of Use

Choosing the right optimizer is crucial for effective training. AdamW is an advanced optimizer that offers improved performance through decoupled weight decay, making it a robust choice for many deep learning applications. Understanding and tuning this hyperparameter can significantly enhance model performance and training efficiency.

4.2.7 Training Process

This section outlines the process for training a Longformer model tailored for masked language modeling tasks. The procedure involves initializing a tokenizer and model configuration, 4.3, preparing the training dataset, and managing the training execution across multiple epochs. The ultimate goal is to save both the trained base model and tokenizer.

```
tokenizer = build_tokenizer(train_files=["pretrain-tokenizer-file.txt"])
config = config_longformer(tokenizer=tokenizer)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = LongformerForMaskedLM(config=config).to(device)
model.resize_token_embeddings(len(tokenizer))

pretrain_dataset = PreTrainDataset.get_with_file(
    tokenizer=tokenizer, file_path="pretrain-dataset-file.txt"
)

num_epochs = 8
optimizer = AdamW(model.parameters(), lr=5e-5)

scaler = GradScaler()
```

Listing 4.3: Example Pretain Components Initialization

A really simple MLM training script, code listing 4.4, follows this logic sequence:

1. Initializes the tokenizer from training files.
2. Configures the model using the built tokenizer.
3. Resizes the token embeddings to match the tokenizer's vocabulary size.
4. Loads and processes the dataset file for pretraining.
5. Sets up the data collator for handling masked language modeling, (masking tokens).

6. Initializes the optimizer (AdamW) and learning rate scheduler.
7. Configures training parameters like batch size, number of epochs, and uses gradient scaling for training with mixed precision, fp32 and fp16.
8. Executes the training across multiple epochs.
9. Saves the trained model and tokenizer to a directory.

```
for epoch in range(num_epochs):
    mlm_probability = 0.38 - (epoch * (0.38 - 0.08) / (num_epochs - 1))
    print(f"Epoch {epoch + 1}/{num_epochs}, MLM Probability: {mlm_probability}")

    data_collator = DataCollatorForLanguageModeling(
        tokenizer=tokenizer,
        mlm=True,
        mlm_probability=mlm_probability,
    )

    train_dataloader = DataLoader(
        pretrain_dataset,
        batch_size=2,
        shuffle=True,
        collate_fn=data_collator,
    )

    scheduler = get_linear_schedule_with_warmup(
        optimizer,
        num_warmup_steps=20,
        num_training_steps=len(train_dataloader) * num_epochs,
    )

    train_epoch(
        model=model,
        train_dataloader=train_dataloader,
        scaler=scaler,
        optimizer=optimizer,
        scheduler=scheduler,
        device=device,
        epoch_n=epoch,
        num_epochs=num_epochs,
    )
```

Listing 4.4: Example Simple Pretrain Logic, with dynamic masking probability

The following components were used in the script, code listing 4.4:

- **LongformerForMaskedLM** from `transformers` for the model architecture.
- **DataCollatorForLanguageModeling** from `transformers` for data preparation.
- **AdamW** and **get_linear_schedule_with_warmup** from `transformers` for optimization.
- **GradScaler** from `torch.cuda.amp` for mixed precision training.
- **DataLoader** from `torch.utils.data` for batching and data loading.

- Local modules for configuration, training, dataset preparation, and tokenizer building.

The procedure successfully saves the trained model and tokenizer ensuring that they are ready to be finetuned. Next a deeper look on epoch level training is provided, inspect code sample 4.5.

Epoch Training Process

1. **Autocasting**: Enables mixed precision for performance efficiency. Computations that benefit from higher precision use `float32`, while others use `bfloat16`.
2. **Batch Processing**: Iterates through batches of data, where each batch is forwarded through the model to compute losses.
3. **Loss Computation**: Uses the provided loss function integral to the model. Loss calculation is essential for backpropagation.
4. **Gradient Scaling**: Applies scaling to the gradients to prevent underflow during backpropagation in mixed precision.
5. **Backward Pass**: Calculates gradients of the model parameters with respect to the loss.
6. **Gradient Clipping**: Limits the magnitude of gradients to a maximum norm to prevent exploding gradient problems.
7. **Optimizer Step**: Updates the model parameters based on computed gradients.
8. **Scheduler Step**: Adjusts the learning rate according to the defined schedule.
9. **Checkpoint Saving**: Periodically saves the model state to facilitate model recovery or further training later.

Each training step modifies the model's weights and internal state, reflecting progression in learning.

```
def train_epoch(
    model: LongformerForMaskedLM,
    train_dataloader: DataLoader,
    scaler: GradScaler,
    optimizer: torch.optim.Optimizer,
    scheduler: torch.optim.lr_scheduler.LambdaLR,
    device: torch.device,
    epoch_n: int,
    num_epochs: int,
):
    model.train()

    total_loss = 0.0
    nan_count = 0
    num_batches = 0

    for step, batch in enumerate(
        tqdm(train_dataloader, desc=f"Epoch {epoch_n+1}/{num_epochs}")
    ):
        batch = {k: v.to(device) for k, v in batch.items()}
        optimizer.zero_grad()

        with autocast(enabled=True, dtype=torch.bfloat16, cache_enabled=True):
            outputs = model(**batch)
            loss = outputs.loss

        if loss is not None and not torch.isnan(loss).any():
            scaler.scale(loss).backward()
            clip_grad_norm_(model.parameters(), max_norm=1.0)
            scaler.step(optimizer)
            scaler.update()

            total_loss += loss.item()
            num_batches += 1
            print(
                f"""Batch {step+1}/{len(train_dataloader)}, Epoch
                {epoch_n+1}/{num_epochs}: Loss = {loss.item():.4f}"""
            )
        else:
            print(f"Skipping batch {step+1} due to NaN loss.")
            nan_count += 1
            optimizer.zero_grad()
            if torch.cuda.is_available():
                torch.cuda.empty_cache()

        del batch, outputs, loss

    avg_loss = total_loss / num_batches if num_batches > 0 else float("nan")
    print(
        f"""Epoch [{epoch_n+1}/{num_epochs}]: Average Loss: {avg_loss:.4f},
        NaNs detected: {nan_count}"""
    )
```

Listing 4.5: Example Epoch Training (Pretrain Phase)

As stated early hyperparameter tuning is really important in model enginenering (model hyper-parameters) but also in model training, a list of training hyperpratameter are listed in the following table.

Hyperparameter	**Description**
Optimizer Function	The algorithm used to adjust the weights of the neural network to minimize the loss function. AdamW was the defined choise.
Learning Rate	The step size used by the optimizer to update the weights during each iteration. A higher learning rate can speed up training but may cause instability, while a lower learning rate can result in more stable training but slower convergence.
Warmup Steps	The number of initial steps during which the learning rate is gradually increased from a low value to the set learning rate. This helps to stabilize training at the beginning.
Batch Size	The number of training samples used in one forward and backward pass. Larger batch sizes can improve the stability of gradient estimates but require more memory.
Masking Probability	The probability of masking input tokens during training.
Masking Probability Decay	The rate at which the masking probability decreases from a maximum to a minimum value over the course of training. Linear decay was used.
Total Number of Epochs	The total number of times the entire training dataset is passed through the network. More epochs can lead to better performance but may also increase the risk of overfitting.

Table 4.3: Pre-Training Hyperparameters

4.3 Finetuning Service

Finetuning a transformer model for this specific task of multi-label sequence classification involves adapting the previous pre-trained transformer to classify sequences (texts).

In this stage the main objective is to make the model capable of classifying project types using the previous knowledge gathered from the pre-training stage.

This is done by adding a classification layer (Fully Connected), also refered to as Linear layer or Dense layer, to the model and training it on labeled data, allowing the model to learn to predict multiple labels for each input sequence.

The process of finetuning involves the following sequence of operations:

- *DataSets Loading*: Load datasets that will be used to adapt the base model.

- *Input Tokenization*: Tokenize the new datasets using the same tokenizer.
- *Training Epochs*: Continue training the model on the datasets to optimize its performance.
- *Finetuned Model*: Achieve a version of the model that is specifically tuned to perform well on its intended tasks.

4.3.1 Redis DataFrames

All data used to train this model was stored both in a Database Management System (DBMS) and in an in-memory database service (Redis), which utilizes a key-value data type. Redis was employed to alleviate stress from the main DBMS, which serves as the operational database for the server, and to expedite data retrieval and execution of rapid operations. Once a substantial dataset size has been achieved for model training, testing, and validation, the engineer can proceed to the fine-tuning phase of the model. An example code sample is provided below on how to load training, validation, and test datasets from Redis along with an identifier of the key and its content, text and labels.

```
async def load_from_redis(
    existing_keys: List[str] | None = None, dataset_type: str = "Train"
) -> Tuple[List[Dict[str, Any]], List[str]]:
    async with Redis(
        host="localhost", port=6379, db=0, decode_responses=True
    ) as redis_client:
        all_keys = await redis_client.keys(f"{dataset_type}:*")
        new_keys = (
            set(all_keys) - set(existing_keys)
            if existing_keys else all_keys
        )
        result = []
        for key in new_keys:
            project_data = await redis_client.hgetall(key)
            labels = list(map(float, project_data.get("labels").split(" ")))
            result.append(
                {"key": key, "text": project_data.get("text"), "labels": labels}
            )
        return result, list(new_keys)
```

Listing 4.6: Example DataSet Loading from a Redis Server

Upgrading the dataset in an asynchronous manner is really easy, and it just requires setting up asynchronous functions to handle data updates efficiently. An example code sample is provided to demonstrate the updating operation.

```
async def update_dataframe(
    keys: List[str], prev_df: pd.DataFrame, type_name: str
) -> Tuple[pd.DataFrame, List[str]]:
    updated_data, new_keys = await load_from_redis(keys, type_name)
    if new_keys:
        updated_df = pd.DataFrame(updated_data)
        return (
            pd.concat([prev_df, updated_df]).drop_duplicates(subset="key"),
            new_keys,
        )
    return prev_df, []
```

Listing 4.7: Example DataSet Updates in an Asynchronous Context from a Redis Server

4.3.2 DataSet and Data Loading

Datasets are constructed in a manner similar to the pre-training phase, however, only annotated datasets should be loaded. Three independent datasets must be initialized before proceeding with any fine-tuning training stage. Have a look at code sample 4.8

```
class FineTuningDataset(Dataset):
    def __init__(self, tokenizer, dataframe: pd.DataFrame, max_len: int):
        self.tokenizer = tokenizer
        self.data = dataframe
        self.text = dataframe["text"].values
        self.targets = list(dataframe["labels"])
        self.max_len = max_len

    def __len__(self) -> int:
        return len(self.data)

    def __getitem__(self, index: int) -> Dict[str, Any]:
        text = self.text[index]
        inputs = self.tokenizer(
            text,
            add_special_tokens=True,
            max_length=self.max_len,
            padding="max_length",
            truncation=True,
            return_tensors="pt",
        )
        input_ids = inputs["input_ids"].squeeze()
        attention_mask = inputs["attention_mask"].squeeze()

        return {
            "input_ids": input_ids,
            "attention_mask": attention_mask,
            "labels": torch.tensor(self.targets[index], dtype=torch.float),
        }
```

Listing 4.8: Example FineTune DataSet linked to a Tokenizer

4.3.3 Loss Calculation and Optimizer

This section briefly outlines the rationale for choosing the loss function and optimizer in training for sequence classification tasks (multi-label). BCEWithLogitsLoss was selected as the loss function, and AdamW was chosen as the optimizer (described in the pretain service). The reason for the loss function choise is provided below.

Cross-Entropy Loss for Multi-Label Classification:

- **Standard Cross-Entropy Loss** is typically utilized for single-label classification tasks where each instance is associated with one label out of many possible options, one example is the MLM training in the pretrain stage, the model predicts one class (token) in a range of possible choises. This function leverages the softmax operation to predict a probability distribution across all classes.

- **Limitation for Multi-Label:** In a multi-label classification context, each instance can be tagged with multiple labels. Standard cross-entropy loss, paired with softmax, is not suitable because softmax assumes that class labels are mutually exclusive—the probabilities must sum to one. Thus, it is inappropriate for scenarios where more than one label may be correct, as softmax compels the model to favor one label over all others.

BCEWithLogitsLoss for Multi-Label Classification:

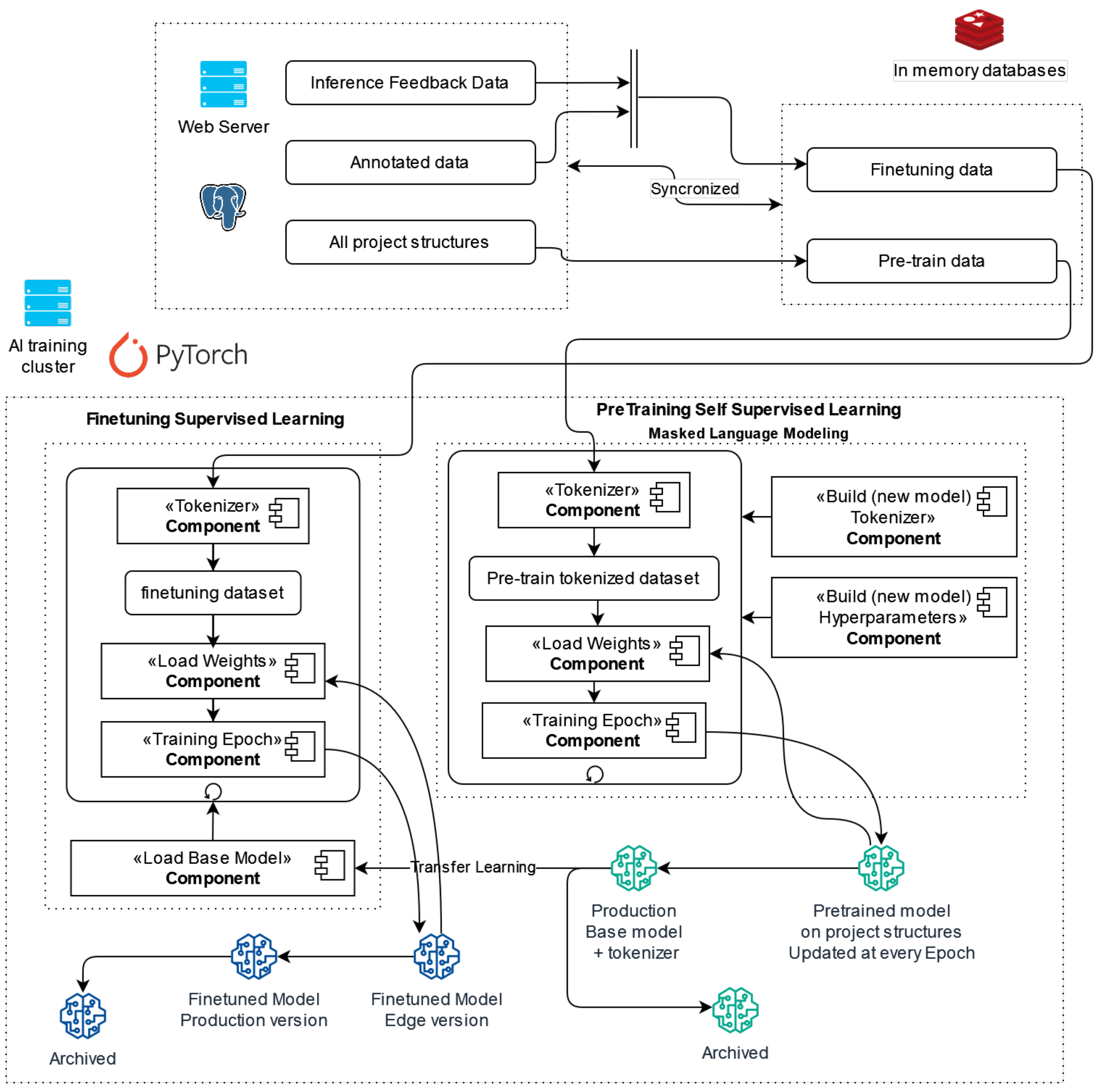
- **Binary Cross-Entropy Loss:** This loss treats each output as a distinct binary classification problem, which is essential for multi-label classification where each label is predicted independently.
- **BCEWithLogitsLoss:** This function is an extension of binary cross-entropy that handles logits directly. It merges a sigmoid activation—which maps logits to probabilities between 0 and 1—with binary cross-entropy loss, allowing for independent prediction of each label.
- **Numerical Stability:** 'BCEWithLogitsLoss' enhances numerical stability by integrating sigmoid activation and loss computation into a single step, preventing issues such as underflow and overflow.
- **Use Case Suitability:** 'BCEWithLogitsLoss' is ideal for multi-label classification as it allows the model to learn to predict each label independently based on its presence or absence, making it suitable for instances tagged with multiple labels.

While cross-entropy loss is effective for single-label classification due to its simplicity, its reliance on softmax renders it unsuitable for multi-label classification where labels are non-exclusive. Conversely, 'BCEWithLogitsLoss' is perfectly suited for such tasks, offering the ability to manage multiple independent binary labels efficiently, ensuring robust model performance in these contexts.

4.3.4 Automatic Model Training

This section provides a brief overview of the model training process. An automated architecture was implemented to manage the flow of data, feedback, and annotations necessary for the model's fine-tuning stage. This architecture operates asynchronously, retrieving data from a Redis server.

The model learns rapidly and adapts effectively to new information. Although this might appear excessive for the specified task of software projects classification, the design targets a more general sequence classification problem. Leveraging available computing resources for training in this automated manner offers several advantages. To visualize the process refer to figure 4.4.



Web Server
Inference Feedback Data
Annotated data
All project structures
Syncronized
In memory databases
Finetuning data
Pre-train data
AI training cluster
PyTorch
Finetuning Supervised Learning
PreTraining Self Supervised Learning
Masked Language Modeling
«Tokenizer» Component
finetuning dataset
«Load Weights» Component
«Training Epoch» Component
«Load Base Model» Component
Pre-train tokenized dataset
«Build (new model) Tokenizer» Component
«Build (new model) Hyperparameters» Component
Transfer Learning
Production Base model + tokenizer
Pretrained model on project structures Updated at every Epoch
Finetuned Model Production version
Finetuned Model Edge version
Archived
Archived

Figure 4.4: Automatic Model Training Diagram

```python
train_keys, valid_keys, test_keys = [], [], []
train_df, valid_df, test_df = pd.DataFrame(), pd.DataFrame(), pd.DataFrame()

iteration = 0
skip_overfitting_checks = 0

last_saved_time = datetime.datetime.now()

while True:
    train_df, new_train_keys = await update_dataframe(
        train_keys, train_df, "Train"
    )
    train_keys.extend(new_train_keys)

    valid_df, new_valid_keys = await update_dataframe(
        valid_keys, valid_df, "Validation"
    )
    valid_keys.extend(new_valid_keys)
    test_df, new_test_keys = await update_dataframe(
        test_keys, test_df, "Test"
    )
    test_keys.extend(new_test_keys)

    num_epochs = calculate_epochs(len(train_df))

    # Initialize Training components, Optimizer, Scheduler, DataLoader (...)
    for epoch in range(num_epochs):
        # Train Epoch

    # (...)
```

Listing 4.9: Example Dataset Loading From redis.

The model undergoes training over a series of epochs, during which it continuously and asynchronously updates its datasets, refer to code listing 4.9. There are two main problems with this:

- Overfitting
- Data Quality Control

Addressing the first issue of overfitting is straightforward, as checks have been implemented to mitigate it. However, the second issue data quality control is more complex. Ensuring the integrity of data annotations and feedback is crucial, and should only be handled by qualified personnel. Training rapidly without revalidating the data could lead to serious issues, particularly concerning security.

4.3.5 DataSet Size Problems

Small annotated datasets pose significant challenges for training effective classification models:

- Limited Learning: The model has fewer examples to learn from, which can lead to poor generalization to new, unseen data.
- Overfitting: With limited data, the model may memorize the training examples rather than learning general patterns, leading to poor performance on different datasets.
- Bias and Variability: Small datasets may not represent the full variability of the real world, introducing bias and reducing the model's robustness.

These issues highlight the need for sufficient annotated data to train reliable and accurate classification models. To address this issue, overfitting checks are required, have a look at code sample 4.10, note that the code should be inside an asynchronous function.

```
if avg_valid_loss > 5 * avg_train_loss:
    skip_overfitting_checks -= 1

    if skip_overfitting_checks <= 0:
        skip_overfitting_checks = 8
        print(
            """Overfitting detected, awaiting for the training
            dataset to double in size..."""
        )
        prev_train_size = len(train_df)
        while len(train_df) < 2 * prev_train_size:
            await asyncio.sleep(60)
            train_df, new_train_keys = await update_dataframe(
                train_keys, train_df, "Train"
            )
            train_keys.extend(new_train_keys)
    else:
        print(
            f"""Overfitting detected, skipping overfitting checks
            {skip_overfitting_checks}/8"""
        )
```

Listing 4.10: Example Overfitting Check, With Dynamic Dataset Updates.

4.3.6 Data Augmentation

Data augmentation is a critical technique used to enhance the performance and accuracy of models, particularly in scenarios where data is scarce or imbalanced. Although it was not employed in this project, data augmentation remains an important concept in situations where data representativity is a concern.

Data augmentation involves generating synthetic data points from existing data through various transformations. This process increases the diversity and quantity of data available, aiding in the development of robust models that generalize better to new, unseen scenarios by simulating a broader range of possible inputs.

This technique is particularly useful during the pretraining stage, which, in our case, already incorporates a substantial dataset (60,000 projects). However, during the finetuning stage where the annotated data is limited (2,000 entries), data augmentation is not effective. This is because augmenting data does not address the fundamental issue of insufficient annotations.

Importance of Data Augmentation

1. **Overcoming Overfitting**: Overfitting occurs when a model learns the details and noise in the training data to an extent that it negatively impacts the performance of the model on new data. Data augmentation increases the size and diversity of the training set, helping reduce overfitting.

2. **Improving Model Robustness**: By introducing variations in the training data, models learn to ignore irrelevant variations in new data, such as noise in the project

structures for our case, markdown files, hashes, ..., data augmentation really deppends on the problem type.

3. **Handling Imbalanced Datasets**: In cases where some classes are underrepresented, data augmentation can balance the dataset by artificially enhancing the minority classes, although this is just theoritically based and was not implemented.

The techniques of data augmentation vary widely depending on the type of data (e.g., images, text, audio) and the specific requirements of the application.

Challenges and Considerations

While beneficial, data augmentation must be applied judiciously:

- **Relevance of Augmentations**: The augmentations should reflect realistic variations, irrelevant modifications can lead to worse model performance.
- **Computational Resources**: More data means more computational overhead, potentially increasing the training time and resource consumption.
- **Balance and Bias**: Care must be taken to ensure that augmentation does not introduce or perpetuate bias in the data.

Data augmentation is a powerful tool in the data scientist's toolkit, providing a means to enhance model training and performance significantly. The role of data augmentation in achieving high-performing, robust models becomes increasingly important. Effective implementation of data augmentation can lead to more accurate, reliable, and fair AI systemsl

4.3.7 Automatic Annotations From LLM

This section explains how the problem of insufficient annotations was addressed. The integration of a local Large Language Model (LLM), such as Code Llama 2 with 70 billion parameters or Llama 3 with 8 billion parameters, provides valuable insights. These models can classify project types based on the structure alone, even though they are not specifically optimized for this task. While capable, these LLMs require a significant amount of computing power and may produce inconsistent results, even with a deterministic configuration using instruction-based models.

With proper prompt engineering, instruction-based models can generate annotations without human supervision. This approach was considered to provide a starting point for model training. Now, with user feedback from inference and subsequent data annotations, the model continuously improves through automatic training. Here is an overview of the process.

- The user asks the model to describe the project structure.
- Assistant reponds.

- User asks specific questions and specifies the way the assistant should respond (true/false) to cast 0/1.
- Assistant reponds, true / false.

The resulting dataset was consistent and reasonably balanced, the overall precision could not be human evaluated.

4.3.8 Training process

The training process does not differ significantly from the previous stage (pretraining the base model).

Here is the simplified logic of training:

1. Transfer the batch data to the specified device.
2. Zero the gradients of the model parameters to prepare for backward propagation.
3. Enable automatic mixed precision via `autocast` for the forward pass, calculating predictions and loss.
4. Scale the loss using `scaler` to maintain numerical stability during backpropagation.
5. Perform a backward pass to compute gradients.
6. Step the optimizer and `scaler` to update model parameters and adjust the `scaler`'s state.
7. Update the learning rate scheduler.
8. Accumulate the loss for averaging at the end of the epoch.

Number of Epochs

The number of epochs follows a linear scaling formula, so that the model updates periodically at constant rates:

$$\text{epochs} = \text{max\_epochs} - \left(\frac{\text{max\_epochs} - \text{min\_epochs}}{\text{max\_size} - \text{min\_size}}\right) \cdot (n - \text{min\_size})$$

The formula adjusts the number of epochs based on the following considerations:

- For dataset sizes greater than or equal to max_size, the function assigns the minimum number of epochs (min_epochs).
- For dataset sizes less than or equal to min_size, the function assigns the maximum number of epochs (max_epochs).
- For dataset sizes within the range of min_size to max_size, the function linearly scales the number of epochs between max_epochs and min_epochs.

Split Distribution

The dataset distribuition is performed by the web server on data annotation. The Web server dynamically assigns one of three categories ('train', 'validation', or 'test') to an element based on the total number of elements. It employs linear adjustments to the distribution ratios with normalization at 10 million elements, ensuring a higher proportion of data is used for training with smaller datasets to prevent overfitting, while allowing more balanced splits as dataset size increases.

The probability of an element being assigned to the 'train' category is calculated as follows:

$$\text{train\_split} = \text{train\_min} + \left(\frac{\text{train\_max} - \text{train\_min}}{\text{threshold}}\right) \times (\text{total\_elements})$$

The probability of an element being assigned to the 'validation' category is calculated as follows:

$$\text{val\_split} = \text{val\_max} - \left(\frac{\text{val\_max} - \text{val\_min}}{\text{threshold}}\right) \times (\text{total\_elements})$$

The probability of an element being assigned to the 'test' category is calculated as follows:

$$\text{test\_split} = 1 - \text{val\_split} - \text{train\_split}$$

- For datasets smaller than 10 million elements, the training split increases from 68% to 84% as the dataset grows, while the validation and test splits decrease correspondingly from 16% to 8%.
- For datasets with 10 million elements or more, the splits are fixed at 84% for training, and 8% each for validation and test.

Training Hyper-parameters

As stated early hyperparameter tuning is really important in model training, a list of training hyperpratameters are listed in the following table.

4.3.9 Model Evaluation

Model evaluation is crucial for understanding the performance of a classification model. It provides a metric (average loss) that helps gauge how well the model predicts the correct labels in a dataset. By evaluating the model, we can:

- Assess Accuracy: Determine how accurately the model is performing on unseen data, and not used to tune hyperparameters (validation).

Hyperparameter	**Description**
Optimizer Function	The algorithm used to adjust the weights of the neural network to minimize the loss function. AdamW was the defined choise.
Learning Rate	The step size used by the optimizer to update the weights during each iteration. A higher learning rate can speed up training but may cause instability, while a lower learning rate can result in more stable training but slower convergence.
Warmup Steps	The number of initial steps during which the learning rate is gradually increased from a low value to the set learning rate. This helps to stabilize training at the beginning.
Batch Size	The number of training samples used in one forward and backward pass. Larger batch sizes can improve the stability of gradient estimates but require more memory.
Overfitting Check	The Logic for identifying and managing overfitting, currently a constant ratio is the detection mechanism.
DataSet Distribution Logic	A change in the maximum possible value of train dataset size, and minimum, as well as the treshold impacts the model generalization capabilites.

Table 4.4: Finetune Training Hyperparameters

- Identify Issues: Spot any potential problems, such as overfitting or underfitting.
- Guide Improvements: Use the evaluation results to make necessary adjustments and improvements to the model.

The evaluation method gives a quantitative measure of the model's effectiveness, ensuring it meets the desired performance standards, for additional details refer to code listing 4.11.

```python
def evaluate_model(
    model: LongformerForSequenceClassification,
    data_loader: DataLoader,
    device: torch.device,
    loss_fct: torch.nn.Module,
) -> float:
    model.eval()

    total_loss = 0
    with torch.no_grad():
        for _, batch in enumerate(data_loader):
            batch = {k: v.to(device) for k, v in batch.items()}
            outputs = model(**batch)
            loss = loss_fct(outputs.logits, batch["labels"])
            total_loss += loss.item()
    return total_loss / len(data_loader)
```

Listing 4.11: Example Model Evaluation on a Validation or Test Set.

4.4 Web Server

This section outlines a comprehensive web service that has been developed to enhance user interaction and support multiple features of the project.

The web service is responsible for:

- **Dataset Management:** It manages all datasets, ensuring they are correctly annotated and integrating inference feedback mechanisms for continual improvement.
- **Data Aggregation from Git Services:** The service is capable of gathering data from various Git services. It constructs project trees that are essential to feed as input to the finetuned model for project classification.
- **Classification Requests:** It handles classification requests, often referred to as inference requests.
- **Integration with Large Language Models (LLM):** The service includes integration with advanced large language models, its capable of processing and understanding large volumes of text-based data efficiently.
- **Additional Features:** The web service is designed to be scalable and adaptable, ready to incorporate additional features as the project evolves.

The developed web service is integral to the project's infrastructure, supporting a wide range of functionalities while ensuring user-friendly and efficient interactions.

4.4.1 Overview

The main components and technologies of the web server will be detailed, along with the logical mechanisms for authentication and the design principles for REST APIs. Additionally, an overall assessment of the web server's service quality is provided.

Technological Stack Introduction

FastAPI, in combination with Uvicorn, were utilized as the foundational libraries for writing the web service. Details on This choices are provided.

Uvicorn, a fast ASGI (Asynchronous Server Gateway Interface) server, serves as the interface for FastAPI.

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.7+ based on standard Python type hints. The key features of FastAPI include automatic generation of interactive API documentation, data validation based on Python type hints using Pydantic, and support for asynchronous programming using asyncio. FastAPI is designed to be easy to use and learn, while also being production-ready and highly performant.

FastAPI's asynchronous capabilities allowed the write of non-blocking code that can handle many simultaneous connections efficiently. This is particularly important for modern web applications that often require high concurrency. Key Points on Asynchronous Programming in FastAPI:

- Concurrency and Efficiency: By leveraging asynchronous programming, FastAPI handles many requests concurrently without waiting for I/O operations (like database queries, external API calls, etc.) to complete. This is managed by Python's asyncio library, which provides an event loop for executing asynchronous tasks. This is particularly helpful when dealing with subprocesses like git cloning, and other subcommands used in this projects that require network and disk operations.
- Asynchronous Database Operations: Using asynchronous ORM libraries, such as Tortoise ORM, FastAPI performs non-blocking database operations.
- Multiple Runners: With the integration of uvicorn, the use of multiple threads to support parallel processing is quite easy.

ORM Integration

Tortoise ORM is the foundational ORM (Object Relational Mapping) service for this web server, it is an easy-to-use async ORM inspired by Django. It provides a simple way to interact with databases asynchronously. Key Features:

- Asynchronous Support: Tortoise ORM is built from the ground up to be fully asynchronous.
- Data Models: Models are defined using Python classes.
- Schema Generation: Automatically generates database schemas from the models, compatible with pydantic.

In the next section the main entities created and managed with Tortoise will be detailed, explaining the core functionality of each. Postgres was used as the foundational DBMS integrated in the system through the means of this ORM library, described and explained on the Deployment chapter.

4.4.2 Entities

A List of the principal entities present in the web service is provided. This is a very high level overview of every entity role. The modeled database in meant to be performant, data removal is not a particular concern, data should only be updated and key identifiers remain static.

- Repository : Represents a git repository with one of https url or ssh url.
- Project : Represents a specific version of a project that can be annotated.
- Inference : Represents an inference request on a project.
- Feedback : Represents a given feedback on an inference
- Classification : Represents a project that is annotated

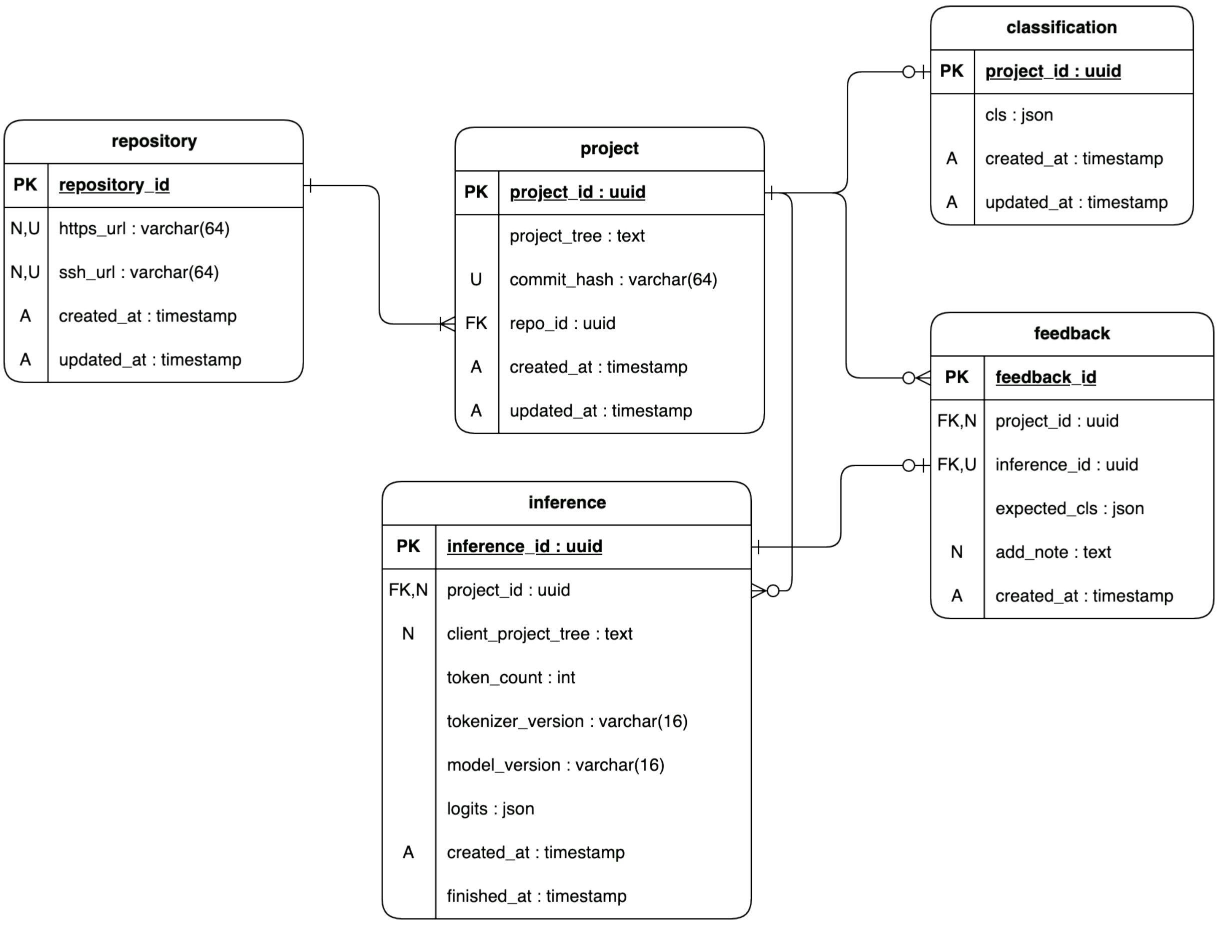


Figure 4.5: Entity Relationship Diagram

4.4.3 Data Validation

Ensuring the integrity and correctness of incoming data in REST API applications is crucial. Pydantic, a Python library, is prominently utilized for data validation due to its efficiency and effectiveness, particularly in conjunction with framework FastAPI framework.

Pydantic plays a crucial role in ensuring data conforms to predefined schemas, improving code quality, and reducing bugs by catching errors early.

- Type annotations for clarity and enhanced type-checking.
- Automatic conversion of complex data types.
- Comprehensive error reporting for improved data quality and debugging.
- Automatic OpenAPI documentation.

Workflow

1. **Request Handling:** The API receives data via requests like GET or POST, which includes query parameters, JSON payloads, or path parameters.

2. **Data Validation:** The incoming data is validated against a predefined schema using Pydantic models before the API processes the request.

3. **Response:** If the data is valid, the API proceeds; otherwise, it returns an error response, 422 (Validation Error).

Benefits of Using Pydantic

- **Efficiency:** Minimizes the need for boilerplate code for manual validation.
- **Clarity:** Clear API documentation through explicit type annotations and models.
- **Security:** Enhances protection against common input-related vulnerabilities.

```python
class LoginSchemaIn(BaseModel):
    # Automatic Regex
    email: EmailStr = Field(
        ..., max_length=255, description="The user's email address."
    )
    hashed_password: str = Field(
        ..., description="Hashed password(client side)."
    )

    @validator("hashed_password")
    def validate_hashed_password(cls, v):
        if len(v) != 64:
            raise ValueError("Invalid hashed password length")
        return v
```

Listing 4.12: Example simple and effective data validation Schema

Data validation is an indispensable aspect of REST API development, crucial for security, reliability, and data correctness.

4.4.4 Endpoints

Endpoints for this service were aggregated into four categories:

- Authentication - These endpoints handle the authentication process, with CSRF double submit protected cookies for JWT authentication, including mechanisms to revoke and refresh tokens.
- Data Management - These endpoints were designed to handle all data operations and administrator or engineer can perform. Mainly related to annotations, inference feedbacks, and projects.
- Inference - These endpoints support the foundational inference service used for the classification task of software project. Tokenizatino is also supported.
- Chat WebSoscket - This endpoint is responsible for establishing a WebSocket connection for real-time interactive chat with a Large Language Model (Local).

4.4.5 Controllers

In the context of REST (Representational State Transfer) API architecture, controllers are essential in managing data flow between the server and clients. They act as intermediaries that process incoming requests, invoke necessary resources or operations, and return responses to the clients. The primary role of controllers is to facilitate communication between the user interface and the application's data services, ensuring efficient data handling and client-server interaction.

Functionality and Operations

Controllers are crucial to the request-handling pipeline within a RESTful service. Their main tasks involve.

- Parsing and validating incoming request data.
- Determining and invoking appropriate actions based on the API's business logic.
- Interacting with models to process data.
- Handling errors and ensuring that appropriate error responses are sent.
- Formatting and sending the response back to the client in a standard format such as JSON.

Design Principles

Designing controllers involved adhering to several key principles that promote maintainability, scalability, and clarity:

1. Single Responsibility
2. Thin Controllers
3. Statelessness
4. Consistency

Efficient controller design significantly impacts the performance and usability of a REST API. By ensuring quick data processing, robust error handling, and clear integration paths, controllers enhance the overall user experience and system reliability.

Controllers are a fundamental component of REST API architecture, responsible for efficiently managing the request-response cycle. Adhering to principles of simplicity, responsibility segregation, and statelessness, effective controller implementation ensures that the API remains robust, scalable, and maintainable.

4.4.6 Environment Isolation

In modern application development, the use of distinct environments for different stages of the software lifecycle is crucial. Two environments were used in the project: a development environment and a production environment.

The development environment is where software engineers design, build, and test new features. It is a sandbox for experimentation and debugging without affecting the live application.

The production environment is the live version of the application, accessible by end users and operating under full load. It is stable, efficient, secure, and isolated from the development environment.

The separation of development and production environments is a fundamental best practice in application development. It ensures that the stability, security, and continuous improvement of applications are maintained, benefiting both the developers and the end-users.

4.4.7 Cross-Origin Resource Sharing(CORS)

Cross-Origin Resource Sharing (CORS) headers control how resources on a web server can be accessed from different domains. Here is a brief explanation of the key CORS headers:

- **Access-Control-Allow-Origin**: Specifies which domains are allowed to access the resources. It can be set to a specific domain, multiple domains, or '*' for all domains.
- **Access-Control-Allow-Methods**: Lists the HTTP methods (e.g., GET, POST, DELETE) that are permitted for accessing the resource. This header is critical in declaring what methods a server will accept from cross-origin requests.
- **Access-Control-Allow-Headers**: Indicates which headers can be included in the actual request. This is used in response to a preflight request to inform the browser about the safe headers that can be used during the actual request.

These headers are fundamental in defining and securing cross-origin interactions by specifying acceptable sources, methods, and headers for requests. Proper CORS configuration was keen to ensure stable and secure deployment of the API and Website.

4.4.8 JWT Authentication

JWT Authentication is a method used to secure communications between clients and servers, employing compact, URL-safe tokens encoded in JSON format. These tokens facilitate token-based authentication, allowing users to access protected routes or resources. The tokens include encoded JSON objects that carry claims representing user properties and other metadata.

Access tokens are issued during the authentication process, granting the bearer the right to access specific resources for a limited period. They are generally short-lived to minimize the security risks in case they are compromised.

Refresh tokens are designed with a longer lifespan and are utilized to request new access tokens once the current access token expires. This arrangement ensures continuous user authentication without the need to repeatedly enter credentials, while maintaining robust security.

4.4.9 Combining JWT with Protected Cookies

Cookies are often employed to securely store JWTs on the client side. Ensuring cookie security involves setting certain flags:

- **HttpOnly:** Makes the cookie inaccessible to client-side scripts, effectively preventing XSS (Cross-Site Scripting) attacks.
- **Secure:** Ensures the cookie is sent only over HTTPS, safeguarding the data during transmission over unsecured connections.
- **SameSite:** This attribute can be set to either strict or lax to mitigate CSRF (Cross-Site Request Forgery) attacks by controlling how cookies are sent with requests from external sites.

CSRF (Cross-Site Request Forgery) Protection

CSRF attacks exploit the trust that a site has in a user's browser, by tricking the user into submitting a request that they did not intend. To counteract CSRF attacks in scenarios using JWTs in cookies, this strategies were implemented:

- **CSRF Tokens:** These are unique, secret, and unpredictable values generated by the server and sent to the client, required in subsequent requests to validate the request's legitimacy.
- **Double Submit Cookie:** A CSRF protection method where the cookie value is sent back to the server not only as a cookie but also as a request parameter, with the server then comparing both values.

Combining JWT authentication with protected cookies and CSRF protection mechanisms provides a comprehensive security framework. It preserves the integrity and confidentiality of the authentication tokens and protects against common web vulnerabilities, thereby securing user data and authentication credentials effectively.

4.4.10 Asynchronous Subprocesses

In modern web development, managing asynchronous operations efficiently is crucial for building scalable and responsive applications.

Asyncio provides support for creating and managing sub-processes. It can handle I/O-bound and CPU-bound operations by delegating intensive tasks to sub-processes, thereby freeing up the main event loop to continue processing other non-blocking tasks.

This is especially beneficial in a web server environment where handling multiple requests efficiently is critical.

This was particularly useful for subprocesses such as git commands and project tree builds.

4.4.11 Parallelism and Concurrency

This section briefly outlines the importance of this two concepts and how they were used in the web server:

- Concurrency refers to the ability of an application to deal with multiple tasks at the 'same' time. In web programming, this means that a server can handle requests from different users concurrently, without necessarily completing one request before starting another. This is crucial for web servers, which might serve thousands of users simultaneously. Concurrency allows the program to make progress on many tasks, often by switching tasks as conditions dictate (like waiting for data to be fetched from a database).
- Parallelism, on the other hand, refers to actually performing multiple operations at the same time. This can be achieved by using multiple processors or cores. In the context of web programming, parallelism would mean that a server uses multi-threading or multi-processing to handle different requests at the exact same moment, thus speeding up processing by distributing tasks across different CPUs.

Concurrency is about dealing with many things at once (often through task switching), parallelism is about doing many things at once (simultaneously running separate processes or threads). Both are used to improve the efficiency and responsiveness of web applications, allowing them to scale and handle multiple user requests smoothly.

For this project, the use of both concurrency and parallelism was particularly important due to the need for AI model inference, which is a slow operation. This combination allows the web service to be performant, as it can respond to and manage other tasks while a user awaits inference results.

4.4.12 Thread-Safe Calls With Locks

In multi-threaded software development, maintaining data integrity and consistency across threads is essential. Thread-safe calls are techniques that prevent conflicts and errors when multiple threads interact with shared resources. One prevalent method to achieve thread-safety is by using locks.

A lock is a synchronization tool that allows only one thread at a time to access a designated portion of code or data. It serves to prevent data corruption or unpredictable behavior arising from concurrent modifications by multiple threads.

```
async def __lock_file_checker(lock_path: str):
    while os.path.exists(lock_path):
        await asyncio.sleep(1)

```

```python

async def __sleep_while_lock_exists(lock_path: str, timeout: int = 60):
    try:
        await asyncio.wait_for(__lock_file_checker(lock_path), timeout)

    except asyncio.TimeoutError:
        if os.path.exists(lock_path):
            os.remove(lock_path)
            print("Timeout reached and lock file was deleted.")
```

Listing 4.13: Example Lock File Checker.

Refer to code listing 4.13 for a basic implementation of a lock file checker.

```python
async def build_tree(commit: str, repo_id: str) -> str | None:
    repo_path = f'{os.getenv("REPO_CLONE_PATH", "/data/projects")}/{repo_id}'
    lock_path = f"{repo_path}/.lock"

    try:
        await __sleep_while_lock_exists(lock_path)

        with open(lock_path, "x") as _:
            print(f"Lock file created on repository {repo_id}")  # log

        await __checkout(repo_path, commit)

        # build project tree
        # (...)

    finally:
        if os.path.exists(lock_path):
            os.remove(lock_path)
            print(f"Lock file deleted on {repo_path}")  # log
```

Listing 4.14: Example Safe Call With a Lock.

Refer to code listing 4.13 for a basic implementation of a 'safe' call in sharing information over concurrent calls. This was particularly important to prevent modifications to software files (git updates) while another thread is performing a tree build.

```python
async def on_startup():
    global app

    # (...)
    redis_client = await get_redis_client()

    lock = redis_client.lock("dataset_init_lock", timeout=30)
    async with lock:
        if not await redis_client.exists("datasets_initialized"):
            await init_datasets()
            await redis_client.set("datasets_initialized", "true")

    # (...)
```

Listing 4.15: Example Lock on Redis Datasets Initialization.

Refer to code listing 4.15 for an example on how to safely initiliaze a component, when the server starts (on_startup function) it is executed by multiple services, the lock ensures it is safely executed by a single service only.

How Do Locks Work?

Locks operate under a simple principle in a multi-threaded environment:

1. **Acquire the Lock:** A thread must acquire a lock before entering a critical section where shared resources are accessed. If the lock is held by another thread, the requesting thread will block until the lock is available.

2. **Execute the Critical Section:** With the lock acquired, the thread has exclusive access to the shared resources, eliminating concurrent modifications.

3. **Release the Lock:** After executing the critical section, the thread releases the lock, allowing other threads to acquire it and access the shared resources.

Advantages

- **Prevents Data Corruption:** By ensuring exclusive access to resources, locks maintain data integrity.

- **Simple to Implement:** Locks are a straightforward mechanism and are easy to implement.

Disadvantages

- **Potential for Deadlocks:** Improper use of locks can lead to deadlocks, where two or more threads wait indefinitely for each other to release locks.

- **Performance Overhead:** Locks can introduce significant performance overhead by reducing concurrency and forcing threads to wait.

Thread-safe calls with locks are vital for ensuring proper function of multi-threaded applications, protecting against data corruption and inconsistencies. While locks are an effective tool for synchronization, they must be used carefully to avoid introducing deadlocks and other performance issues.

4.4.13 Secure WebSockets

WebSockets were used to enable communication between the client and the local LLM in this project. They are efficient and an excellent choice for chat applications.

Secure WebSockets (WSS) is an advanced communication protocol that enhances the security of data exchange between a client (such as a web browser) and a server over the web. It is an extension of the WebSocket protocol, which facilitates real-time data transfer but with a focus on providing a secure transmission channel.

Before delving into Secure WebSockets, it's important to understand the basic WebSocket protocol. WebSockets provide a way to open a bi-directional communication session between the client and server. This allows data to be sent and received at any time without the need for the client to initiate a request, as required by traditional HTTP connections. The WebSocket protocol enables applications like live notifications, interactive games, and real-time data feeds.

While WebSockets make it efficient to handle real-time data, they do not inherently include mechanisms for encryption, which exposes data to potential interception and tampering. Secure WebSockets address this security gap by encrypting the data transmitted during the session.

Secure WebSockets, denoted by the protocol identifier `wss://` (as opposed to `ws://` for insecure WebSockets), operate similarly to HTTPS. They layer the WebSocket communication over TLS (Transport Layer Security), providing confidentiality, integrity, and authentication. Here's a step-by-step look at how WSS functions.

1. Connection Establishment

2. TLS Handshake

3. Real Time Bidirectional Data Transfer

Secure WebSockets (WSS) are crucial for applications requiring secure real-time data communication. By leveraging TLS, WSS ensures that data exchanged over WebSockets remains private and intact, providing a robust solution for developers to implement real-time, interactive, and secure web applications. As real-time data becomes increasingly prevalent in modern applications, the importance of protocols like Secure WebSockets continues to grow.

4.4.14 DataSet Management

The web service is responsible for dataset management, including handling data annotations and inference feedback. It automatically assigns items to training, testing, and validation sets in a manner that is persistent, safely isolated, and correctly distributed. Additionally, the web service manages a Redis database to share data with training components without affecting the operational functionality of the main DBMS.

4.4.15 AI Model Integration

The integration of the AI model into the web server is crucial for the system's operation. This process involves using an efficient architecture that supports asynchronous operations, allowing the server to handle multiple services concurrently while maintaining system stability. To avoid memory issues, a lock mechanism ensures that one thread only processes an inference request at a time, although the server can perform other tasks asynchronously. Given that each inference request uses up to 800MB of (V)RAM, the number of parallel inference requests can be optimized by using the lock and using multiple processes.

Torch Compile

torch.compile is the latest method to speed up your PyTorch code, torch.compile makes PyTorch code run faster by JIT-compiling PyTorch code into optimized kernels, all while requiring minimal code changes.

Integration Example

```
class InferenceModel:
    _instance = None
    inference_lock = asyncio.Lock()

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    tokenizer = AutoTokenizer.from_pretrained(model_path)
    model = LongformerForSequenceClassification.from_pretrained(
        model_path, num_labels=4
    ).to(device)
    model = torch.compile(model)

    def __new__(cls):
        if cls._instance is not None:
            return cls._instance

        return super(InferenceModel, cls).__new__(cls)

    def __pre_process(self, project_tree: str) -> str:
        def is_ascii(s):
            return re.match(r"^[\x00-\x7F]+$", s) is not None

        # (...)

        return project_tree


    async def __process(self, project_tree: str):
        async with self.inference_lock:
            print("Processing Inference")
            self.model.eval()
            with torch.no_grad():
                inputs = self.tokenizer(
                    self.__pre_process(project_tree),
                    return_tensors="pt",
                    padding=True,
                    truncation=True,
                    max_length=4096,
                ).to(self.device)
                token_count = inputs["input_ids"].size(1)
                outputs = self.model(**inputs)
                logits = outputs.logits
            return torch.sigmoid(logits).cpu().numpy(), token_count

    async def process(self, project_tree: str):
        try:
            logits_array, token_count = await self.__process(project_tree)
            class_labels = ["backend", "frontend", "mobile", "has_api"]
            logits_dict = {
                class_labels[i]: float(logits_array[0, i])
                for i in range(logits_array.shape[1])
            }

            return {
                "token_count": token_count,
                "current_model_version": self.model._version,
                "logits": logits_dict,
            }
        except Exception as e:
            print(f"An error occurred during the processing: {str(e)}")
            raise ValueError("An error occurred during the processing.")

    @classmethod
    def get(cls):
        return cls()
```

Listing 4.16: Example inference integration on the web server.

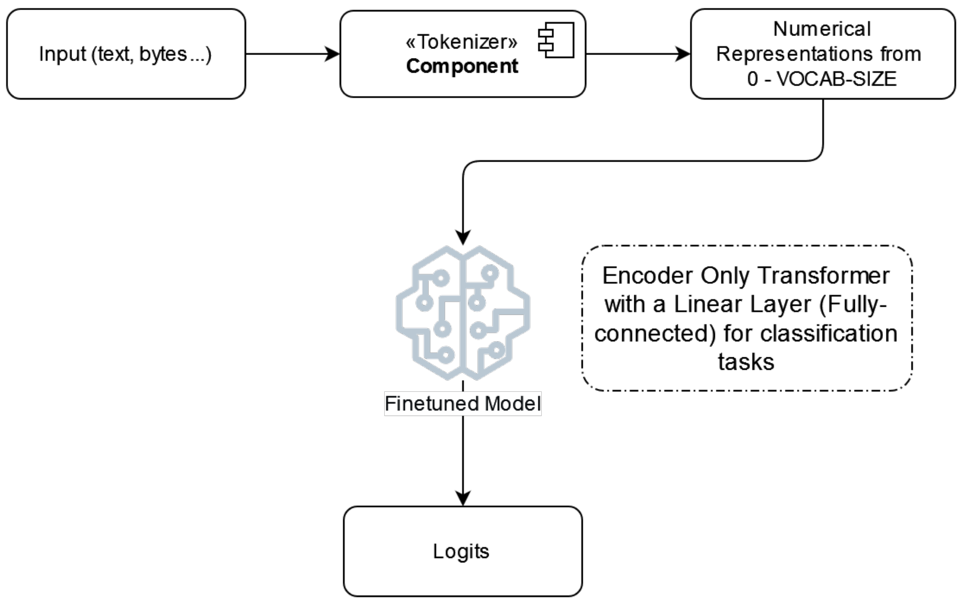

Input (text, bytes ...)
«Tokenizer»
Component
Numerical Representations from 0 - VOCAB-SIZE
Encoder Only Transformer with a Linear Layer (Fully-connected) for classification tasks
Finetuned Model
Logits

Figure 4.6: Inference Process Diagram.

1. **Class Initialization**:

   - `InferenceModel` is a singleton class, meaning only one instance of the model exists at any time.
   - The model and tokenizer are loaded and moved to the appropriate device (GPU or CPU).
   - `torch.compile` is used to optimize the model for faster inference.

2. **Asynchronous Processing**:

   - The class includes a lock to ensure that only one inference request is processed at a time, preventing memory issues.

- The `process` method handles input data, processes it through the model, and returns the results asynchronously.

This setup, code listing 4.16, ensures that the web server can efficiently and safely handle AI model inferences, making the best use of available resources, the process can be visualized in figure 4.7.

4.4.16 Inference Process

This section provides a brief overview of how to request inference calls. There are two methods available:

- Call the API with a Git repository URL.
- Call the API with a client-built project tree, as shown in code listing 4.17.

Figure 4.7 presents the web interface for calling the API using a Git repository.

```bash
#!/usr/bin/env bash

if [ "$#" -ne 1 ]; then
    echo "Usage: $0 <directory>"
    exit 1
fi

directory=$1

cd "$directory" || exit

temp_file=$(mktemp)
eza -TSBF --git-ignore -I '*migrations*|*locale*|*fonts*|*docs*|*.ttf|*.woff|\
*.woff2|*.md|*.png|*.gif|*.jpg|*.svg|*.pdf|*.jpeg|*.mp4|*.mkv|*.webp|\
*translation*|*learn*|*.mdx|*languages*|' | grep -vE '[0-9]{2}' > "$temp_file"

url="https://apiv1.project-cls.0xdbff.dev/inference"

raw_text=$(cat $temp_file)
json_payload=$(jq -n --arg key "$raw_text" '{client_project_tree: $key}')

echo "$json_payload"

response=$(curl -s -L -X POST --max-time 120 "$url" -H \
"Content-Type: application/json" -d "$json_payload")

echo "Response from server: $response"
rm "$temp_file"
```

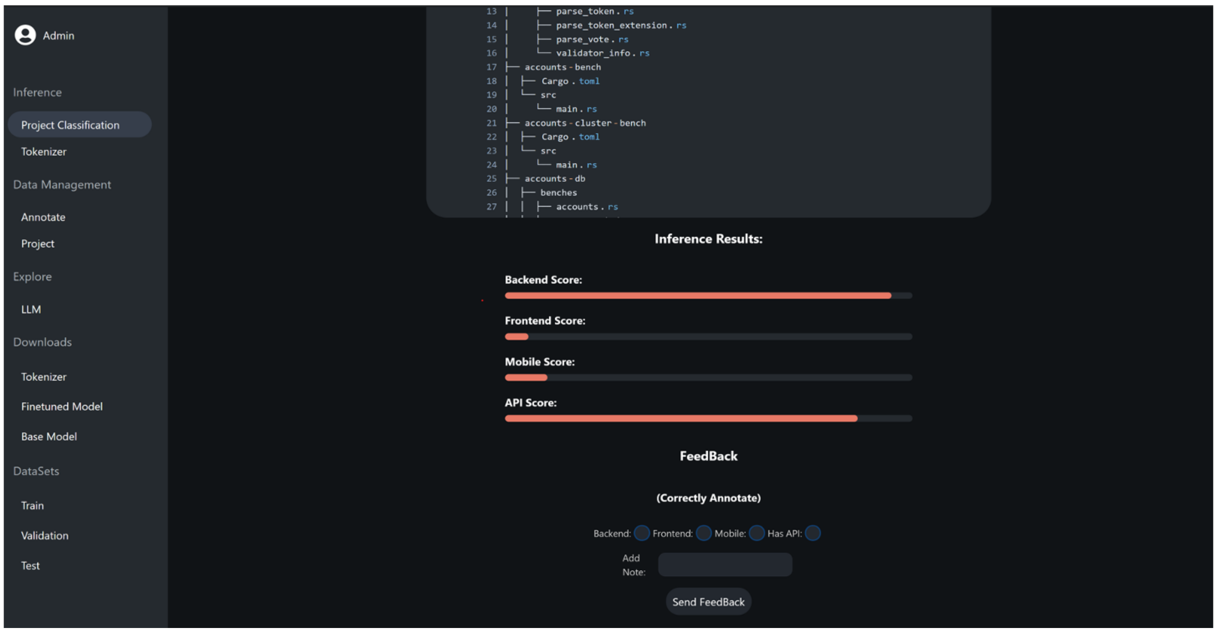
Listing 4.17: Example Inference request, with a client built tree.

Code listing 4.17 represents an example of how to create an inference request in a client by just building a project tree and creating a post request to the API.

```json
{
   "inference_id":"fc6838bd-3567-4c88-ac8a-1d43aa9fcc5a",
   "token_count":305,
   "current_model_version":"1",
   "logits":{
      "backend":0.9478213787078857,
      "frontend":0.05160602554678917,
      "mobile":0.16283629834651947,
      "has_api":0.9419170618057251
   },
   "created_at":"2024-06-04T14:36:48.683165Z",
   "project_id":null
}
```

Listing 4.18: Example inference reponse.

Code listing 4.18 demonstrates an example response for an inference request.



```
│       ├── parse_token.rs
│       ├── parse_token_extension.rs
│       ├── parse_vote.rs
│       └── validator_info.rs
├── accounts-bench
│   ├── Cargo.toml
│   └── src
│       └── main.rs
├── accounts-cluster-bench
│   ├── Cargo.toml
│   └── src
│       └── main.rs
├── accounts-db
│   ├── benches
│   │   ├── accounts.rs
```

Figure 4.7: Inference Request Web Interface

4.4.17 Scripts

This section provides a concise overview of the primary utility scripts developed to support the web service:

- **LLM Annotation** - This script performs automatic annotations on data by utilizing automated prompt engineering and calling the associated LLM service. It simulates user requests to interact with the web service.
- **Best Projects from GitHub** - This script gathers top projects from GitHub and related information by using the Github API. It subsequently calls the web service to insert these projects into the database, which will be used for pretraining.
- **Data Migrations, Backups, and Restoration** - This scripts ensures the seamless migration, backup, and restoration of data, maintaining data integrity and availability.

4.4.18 LLM Integration

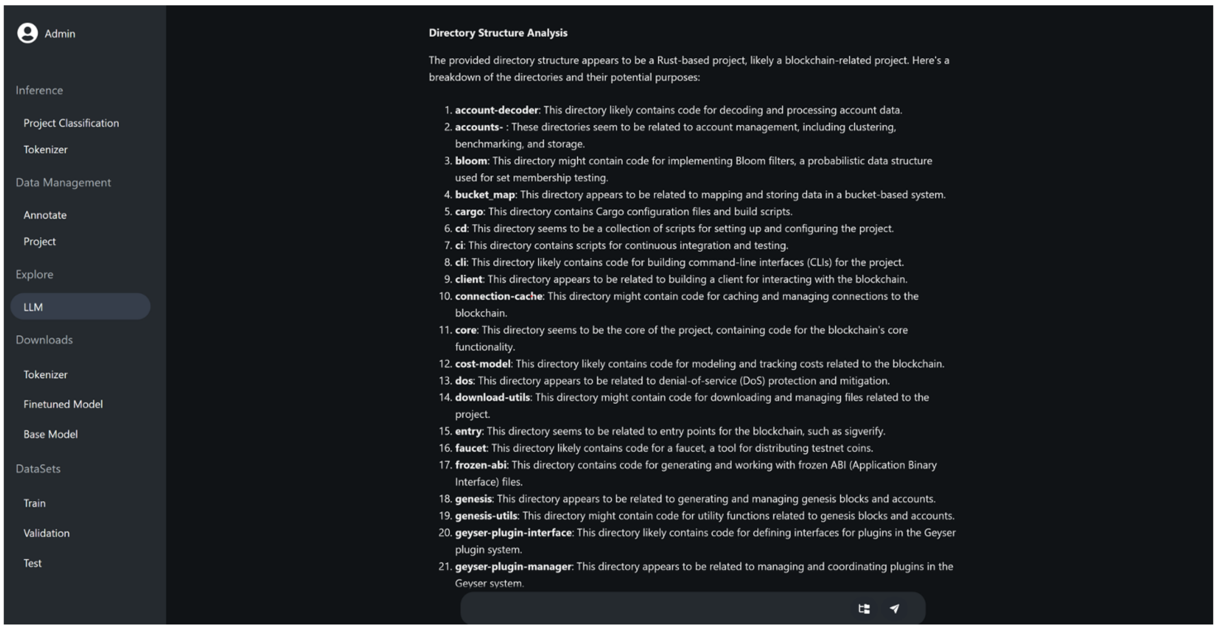
This section briefly details the integration of a Large Language Model (LLM) into the web service, inspect code listing 4.19 to better analyse the process.

1. A websocket endpoint was defined to initiate websocket connections.
2. A service with an instruction model is properly loaded and ready to initiate connections.
3. The instruct model service is properly called by using an OpenAI compatible API, meaning this local LLM can be updated to use ChatGPT.
4. A chat history is defined and tied to the lifetime of the websocket. Note that for production, connection limits and persistent chat history should be employed in the project. Currently, this is just for experimentation.
5. Finally the client has bidirectional communication with the server supporting the conversation with the llm.

```
@router.websocket("/chat/llm/wss")
async def websocket_endpoint(
    websocket: WebSocket,
    authorize: AuthJWTBearer = Depends(auth_dep),
    redis_client=Depends(get_redis_client),
):
    await websocket.accept()
    _ = await validate_access_token(
        authorize, redis_client, validate_sub_with_internal_id=True
    )

    # Connection limit required for production
    conn_chat_history = []

    # (Local) LLM Server with OpenAI compatible API
    # currently using llama 8b or code Llama 2 34b (system not assistant as the
    # response user (openBuddy instruct))
    url = "http://127.0.0.1:5000/v1/chat/completions"
    headers = {"Content-Type": "application/json"}

    try:
        while True:
            input_data = json.loads(await websocket.receive_text())

            if "content" in input_data:
                msg = input_data["content"]
            else:
                raise ValueError("Invalid input data")

            conn_chat_history.append({"role": "user", "content": msg})

            data = {
                "mode": "instruct",
                "stream": True,
                "messages": conn_chat_history,
                "max_tokens": 4096,
                "temperature": 0.7,
                "top_p": 0.1,
            }

            try:
                stream_response = requests.post(
                    url, headers=headers, json=data, stream=True
                )
                stream_response.raise_for_status()
                client = sseclient.SSEClient(stream_response)

                assistant_message = ""
                for event in client.events():
                    payload = json.loads(event.data)
                    if (
                        "choices" in payload
                        and len(payload["choices"]) > 0
                        and "delta" in payload["choices"][0]
                    ):
                        chunk = payload["choices"][0]["delta"]["content"]
                        await websocket.send_text(chunk)
                        assistant_message += chunk

                conn_chat_history.append(
                    {"role": "assistant", "content": assistant_message}
                )

            except requests.RequestException as e:
                print("Failed to connect to server or error in stream: ", e)
            except json.JSONDecodeError:
                print("Error decoding the JSON response from the server")

            await websocket.send_text("|wss_complete_response|")

    except WebSocketDisconnect:
        print("Client disconnected")
```

Listing 4.19: Example websocket endpoint for LLM chatting.

Refer to figure 4.8 and 4.9 to visualize an interaction with the LLM.

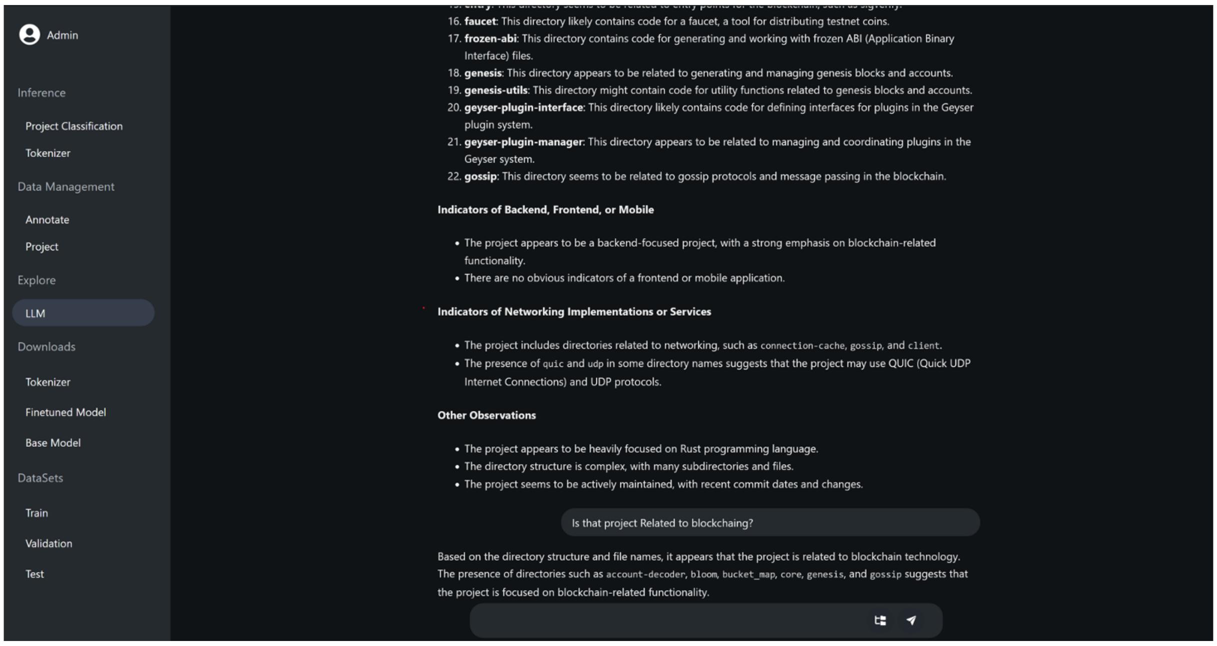


Directory Structure Analysis

The provided directory structure appears to be a Rust-based project, likely a blockchain-related project. Here's a breakdown of the directories and their potential purposes:

1. **account-decoder**: This directory likely contains code for decoding and processing account data.
2. **accounts-** : These directories seem to be related to account management, including clustering, benchmarking, and storage.
3. **bloom**: This directory might contain code for implementing Bloom filters, a probabilistic data structure used for set membership testing.
4. **bucket_map**: This directory appears to be related to mapping and storing data in a bucket-based system.
5. **cargo**: This directory contains Cargo configuration files and build scripts.
6. **cd**: This directory seems to be a collection of scripts for setting up and configuring the project.
7. **ci**: This directory contains scripts for continuous integration and testing.
8. **cli**: This directory likely contains code for building command-line interfaces (CLIs) for the project.
9. **client**: This directory appears to be related to building a client for interacting with the blockchain.
10. **connection-cache**: This directory might contain code for caching and managing connections to the blockchain.
11. **core**: This directory seems to be the core of the project, containing code for the blockchain's core functionality.
12. **cost-model**: This directory likely contains code for modeling and tracking costs related to the blockchain.
13. **dos**: This directory appears to be related to denial-of-service (DoS) protection and mitigation.
14. **download-utils**: This directory might contain code for downloading and managing files related to the project.
15. **entry**: This directory seems to be related to entry points for the blockchain, such as sigverify.
16. **faucet**: This directory likely contains code for a faucet, a tool for distributing testnet coins.
17. **frozen-abi**: This directory contains code for generating and working with frozen ABI (Application Binary Interface) files.
18. **genesis**: This directory appears to be related to generating and managing genesis blocks and accounts.
19. **genesis-utils**: This directory might contain code for utility functions related to genesis blocks and accounts.
20. **geyser-plugin-interface**: This directory likely contains code for defining interfaces for plugins in the Geyser plugin system.
21. **geyser-plugin-manager**: This directory appears to be related to managing and coordinating plugins in the Geyser system.

Figure 4.8: LLM Interaction 1



15. **entry**: This directory seems to be related to entry points for the blockchain, such as sigverify.
16. **faucet**: This directory likely contains code for a faucet, a tool for distributing testnet coins.
17. **frozen-abi**: This directory contains code for generating and working with frozen ABI (Application Binary Interface) files.
18. **genesis**: This directory appears to be related to generating and managing genesis blocks and accounts.
19. **genesis-utils**: This directory might contain code for utility functions related to genesis blocks and accounts.
20. **geyser-plugin-interface**: This directory likely contains code for defining interfaces for plugins in the Geyser plugin system.
21. **geyser-plugin-manager**: This directory appears to be related to managing and coordinating plugins in the Geyser system.
22. **gossip**: This directory seems to be related to gossip protocols and message passing in the blockchain.

Indicators of Backend, Frontend, or Mobile

- The project appears to be a backend-focused project, with a strong emphasis on blockchain-related functionality.
- There are no obvious indicators of a frontend or mobile application.

Indicators of Networking Implementations or Services

- The project includes directories related to networking, such as `connection-cache`, `gossip`, and `client`.
- The presence of `quic` and `udp` in some directory names suggests that the project may use QUIC (Quick UDP Internet Connections) and UDP protocols.

Other Observations

- The project appears to be heavily focused on Rust programming language.
- The directory structure is complex, with many subdirectories and files.
- The project seems to be actively maintained, with recent commit dates and changes.

Is that project Related to blockchaing?

Based on the directory structure and file names, it appears that the project is related to blockchain technology. The presence of directories such as `account-decoder`, `bloom`, `bucket_map`, `core`, `genesis`, and `gossip` suggests that the project is focused on blockchain-related functionality.

Figure 4.9: LLM Interaction 2

The process of LLM chatting through the use of websockets, high level overview, can be inspected in figure 4.10

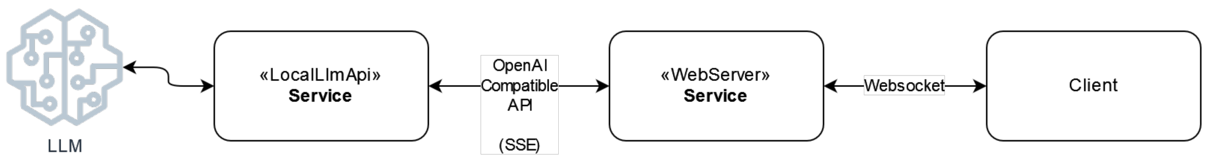


Figure 4.10: LLM Chatting Process Overview.

4.4.19 Unit and Integration Tests

Unit and integration tests are vital for the development and maintenance of web servers. Unit tests focus on individual components, ensuring that each function works correctly in isolation, which helps in early detection and resolution of bugs. Integration tests, on the other hand, validate the interactions between different components of the system, confirming that they work together as intended. The implementation of these tests significantly enhances code quality, minimizes errors during deployment, and ensures a seamless user experience. Additionally, rigorous testing reduces downtime and maintenance costs, providing assurance that updates will not disrupt existing functionalities.

However since the main goal of this internship isn't the build of the web server this tests were not deply explored, only basic examples were coded to give a fundamental and basic start to this critical aspect of automated testing.

4.5 Website

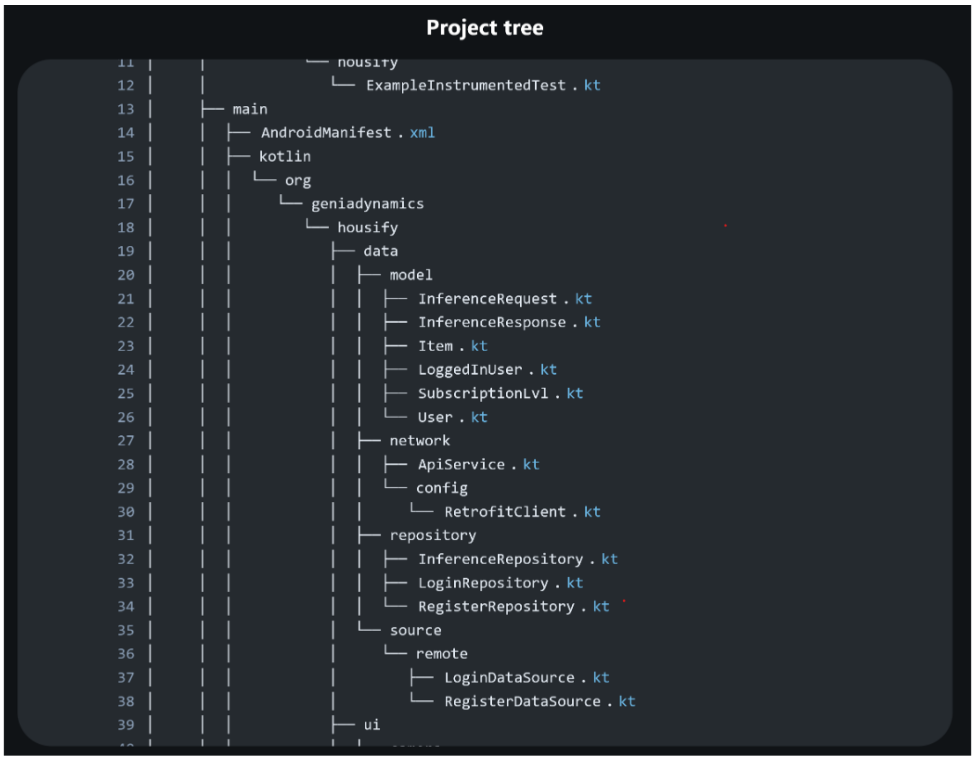
This section briefly outlines the website importance, design, and functionality considerations.

The main use cases/pages of the website are:

- **Authenticate:** Allows users to login to access the website's protected features.
- **Project Classification:** Enables users to request a classification for their software project.
- **Feedback on Inference or Annotation:** Provides a platform for users to give feedback on the model's inference or to annotate projects manually.
- **Tokenizer Visualization:** Allows users to visualize the tokenization process, which is how the model breaks down text into smaller units for analysis.
- **LLM Integration:** Integrates a Large Language Model (LLM) for project structure analysis, chatting, and comparison. This feature allows users to interact with the LLM to gain insights into their projects or to compare different projects.

Tokenizer Visualization

The tokenizer visualization feature allows users to see how the data transitions from the project's tree structure to the token IDs that the model processes. The model only processes token IDs. Refer to Figure 4.11 to visualize the original project tree.

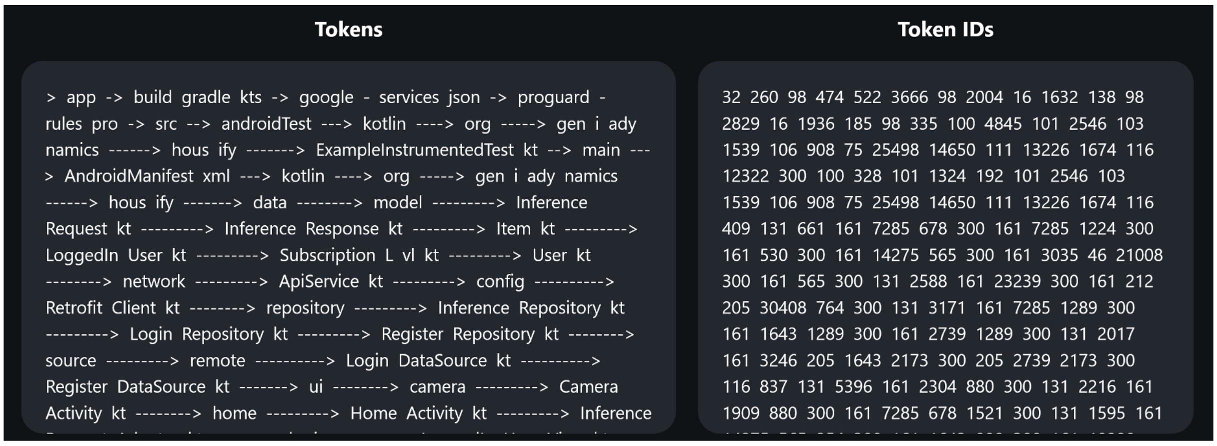


Project tree

```
│       │               └── housify
│       │                   └── ExampleInstrumentedTest.kt
│       ├── main
│       │   ├── AndroidManifest.xml
│       │   ├── kotlin
│       │   │   └── org
│       │   │       └── geniadynamics
│       │   │           └── housify
│       │   │               ├── data
│       │   │               │   ├── model
│       │   │               │   │   ├── InferenceRequest.kt
│       │   │               │   │   ├── InferenceResponse.kt
│       │   │               │   │   ├── Item.kt
│       │   │               │   │   ├── LoggedInUser.kt
│       │   │               │   │   ├── SubscriptionLvl.kt
│       │   │               │   │   └── User.kt
│       │   │               │   ├── network
│       │   │               │   │   ├── ApiService.kt
│       │   │               │   │   └── config
│       │   │               │   │       └── RetrofitClient.kt
│       │   │               │   ├── repository
│       │   │               │   │   ├── InferenceRepository.kt
│       │   │               │   │   ├── LoginRepository.kt
│       │   │               │   │   └── RegisterRepository.kt
│       │   │               │   └── source
│       │   │               │       └── remote
│       │   │               │           ├── LoginDataSource.kt
│       │   │               │           └── RegisterDataSource.kt
│       │   │               ├── ui
```

Figure 4.11: Initial Input (Project Tree)

The tokenization process occurs after the original input is preprocessed to better generalize the information. Once preprocessed, the data is fed into the tokenizer, which converts the input into tokens and finally into their corresponding token IDs. Refer to Figure 4.12 to better visualize this process.



Tokens

```
> app -> build gradle kts -> google - services json -> proguard -
rules pro -> src --> androidTest ---> kotlin ----> org -----> gen i ady
namics ------> hous ify -------> ExampleInstrumentedTest kt --> main ---
> AndroidManifest xml ---> kotlin ----> org -----> gen i ady namics
------> hous ify -------> data --------> model ---------> Inference
Request kt ---------> Inference Response kt ---------> Item kt --------->
LoggedIn User kt ---------> Subscription L vl kt ---------> User kt
--------> network ---------> ApiService kt ---------> config ---------->
Retrofit Client kt --------> repository ---------> Inference Repository kt
---------> Login Repository kt ---------> Register Repository kt -------->
source ---------> remote ----------> Login DataSource kt ---------->
Register DataSource kt -------> ui --------> camera ---------> Camera
Activity kt --------> home ---------> Home Activity kt ---------> Inference
```

Token IDs

```
32 260 98 474 522 3666 98 2004 16 1632 138 98
2829 16 1936 185 98 335 100 4845 101 2546 103
1539 106 908 75 25498 14650 111 13226 1674 116
12322 300 100 328 101 1324 192 101 2546 103
1539 106 908 75 25498 14650 111 13226 1674 116
409 131 661 161 7285 678 300 161 7285 1224 300
161 530 300 161 14275 565 300 161 3035 46 21008
300 161 565 300 131 2588 161 23239 300 161 212
205 30408 764 300 131 3171 161 7285 1289 300
161 1643 1289 300 161 2739 1289 300 131 2017
161 3246 205 1643 2173 300 205 2739 2173 300
116 837 131 5396 161 2304 880 300 131 2216 161
1909 880 300 161 7285 678 1521 300 131 1595 161
```

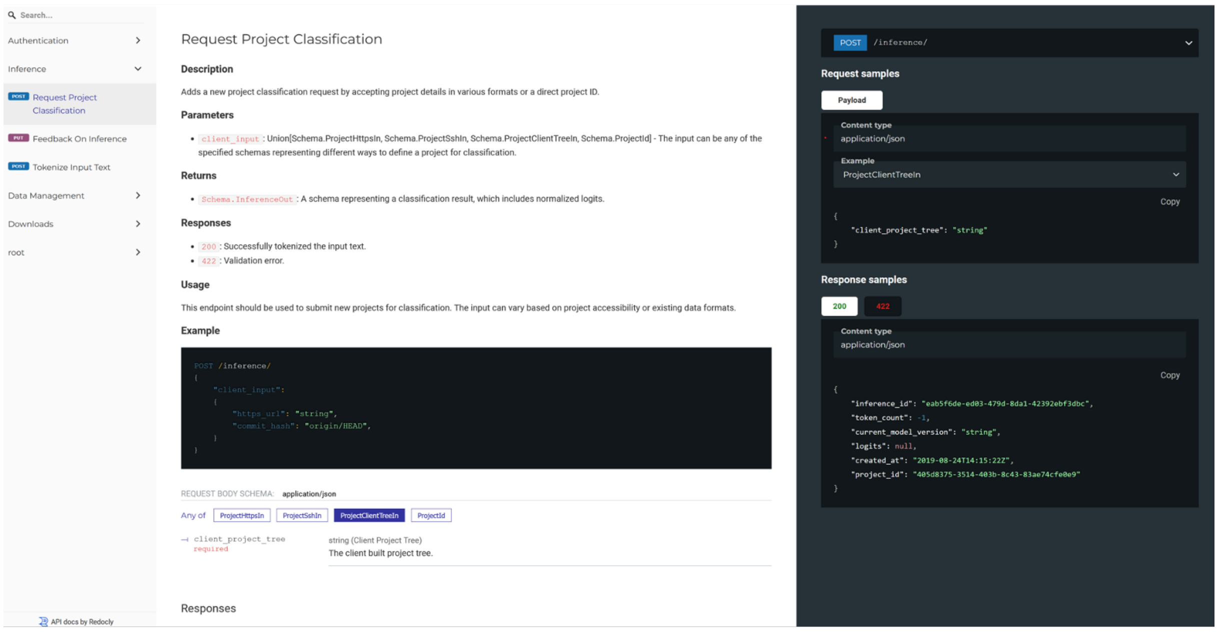
Figure 4.12: Tokenization Process

4.6 Documentation

Documentation serves as an integral component of every software project, ensuring it remains maintainable, readable, and properly functioning.

4.6.1 API Documentation

API documentation is automatically generated following OpenAPI standards and Pydantic schemas. These schemas detail the expected input data for the server and the corresponding responses. All responses must be documented, and typical errors such as HTTP 500 should be avoided. Refer to Figure 4.13 to visualize an api documentation sample.



Request Project Classification

Description

Adds a new project classification request by accepting project details in various formats or a direct project ID.

Parameters

- `client_input` : Union[Schema.ProjectHttpsIn, Schema.ProjectSshIn, Schema.ProjectClientTreeIn, Schema.ProjectId] - The input can be any of the specified schemas representing different ways to define a project for classification.

Returns

- `Schema.InferenceOut` : A schema representing a classification result, which includes normalized logits.

Responses

- `200` : Successfully tokenized the input text.
- `422` : Validation error.

Usage

This endpoint should be used to submit new projects for classification. The input can vary based on project accessibility or existing data formats.

Example

```
POST /inference/
{
    "client_input":
    {
        "https_url": "string",
        "commit_hash": "origin/HEAD",
    }
}
```

REQUEST BODY SCHEMA: application/json

Any of ProjectHttpsIn ProjectSshIn ProjectClientTreeIn ProjectId

`client_project_tree` required — string (Client Project Tree) The client built project tree.

Responses

POST /inference/

Request samples

Payload

Content type: application/json

Example: ProjectClientTreeIn

```
{
    "client_project_tree": "string"
}
```

Response samples

200 422

Content type: application/json

```
{
    "inference_id": "eab5f6de-ed03-479d-8da1-42392ebf3dbc",
    "token_count": -1,
    "current_model_version": "string",
    "logits": null,
    "created_at": "2019-08-24T14:15:22Z",
    "project_id": "405d8375-3514-403b-8c43-83ae74cfe0e9"
}
```

Figure 4.13: Initial Input (Project Tree)

4.6.2 Automatic Code Documentation

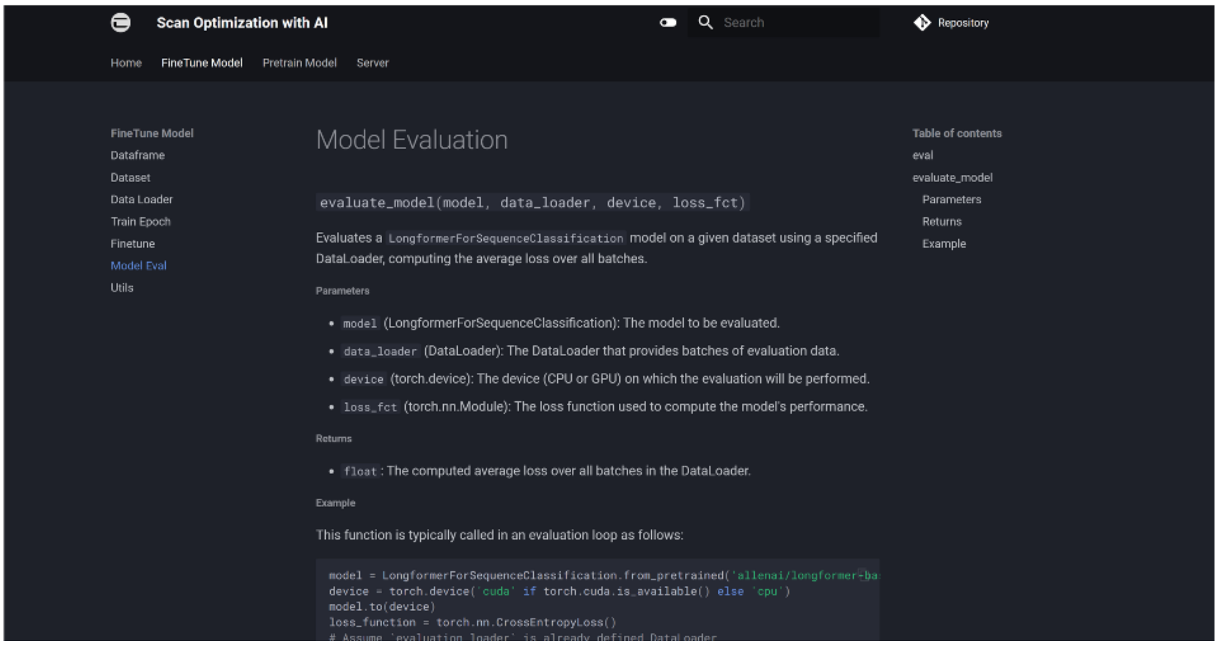
MKDocs is a static site generator tailored for creating project documentation. It transforms markdown files into a deployable HTML site, making it particularly suitable for AI projects that require detailed explanatory documents for various components such as models, algorithms, and data sets.

Documentation is a critical component of Artificial Intelligence (AI) projects, which are often complex and require detailed explanations of the methodologies and technologies employed. MKDocs serves as an effective tool to create well-organized and navigable documentation.

Key Features

- Markdown Support
- Theme Support
- Live Preview
- Search Functionality

MKDocs generates sites that can be easily deployed to numerous hosting services like GitHub Pages, enabling global accessibility and facilitating collaborative developments. Refer to image 4.14 to visualize a sample of project documentation.



Model Evaluation

`evaluate_model(model, data_loader, device, loss_fct)`

Evaluates a `LongformerForSequenceClassification` model on a given dataset using a specified DataLoader, computing the average loss over all batches.

Parameters

- `model` (LongformerForSequenceClassification): The model to be evaluated.
- `data_loader` (DataLoader): The DataLoader that provides batches of evaluation data.
- `device` (torch.device): The device (CPU or GPU) on which the evaluation will be performed.
- `loss_fct` (torch.nn.Module): The loss function used to compute the model's performance.

Returns

- `float` : The computed average loss over all batches in the DataLoader.

Example

This function is typically called in an evaluation loop as follows:

```
model = LongformerForSequenceClassification.from_pretrained('allenai/longformer-ba
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
loss_function = torch.nn.CrossEntropyLoss()
# Assume `evaluation_loader` is already defined DataLoader
```

Figure 4.14: Sample Project Documentation.

5. Data Collection

This chapter outlines the process of data gathering for the pretraining stage. A rich dataset is crucial for developing an effective base model. A high-level overview of the strategy is illustrated in Figure 5.1. The main steps of the data collection process are as follows:

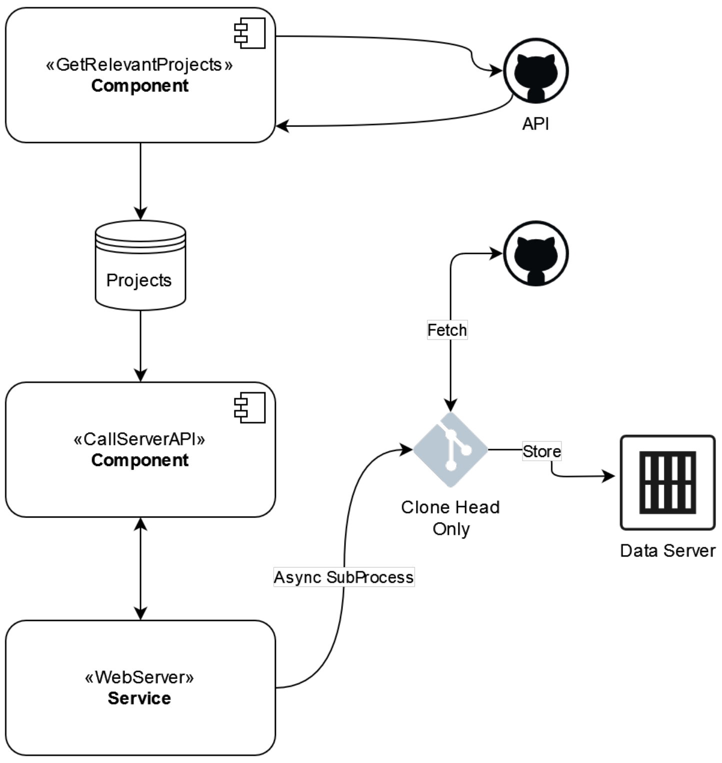


Figure 5.1: Data Gathering Diagram.

1. Gather Relevant Projects from GitHub API: Use the GitHub API to collect relevant projects. This step ensures that the most pertinent and high-quality repositories are selected.

2. Insert the Gathered Projects into the Web Service: Insert the collected projects into the web service. The web service performs cloning and tree building in a safe manner, it's called with the help of a script, simulating a user inserting projects.

3. Gather All Project Trees Available on the Web Server: Collect all the project trees available on the web server. This ensures that the entire structure of each project is captured and ready for further processing.

Pretraining on Small DataSets Problems

- **Limited Vocabulary**: A small dataset may not contain a comprehensive range of vocabulary and language constructs. Transformers require exposure to diverse 'words' to understand text effectively. With a limited dataset, the model may not learn uncommon 'words' or varied linguistic structures, resulting in poor performance.
- **Narrow Context**: Language models benefit from understanding context over long passages. Small datasets often provide limited contextual information, which hampers the model's ability to learn long-range dependencies and nuanced meanings.
- **Overfitting Risk**: When pretrained on small datasets, transformers can easily overfit, meaning they learn the noise and specific patterns of the limited data rather than generalizable language patterns. Overfitting leads to poor performance on new, unseen data as the model fails to generalize beyond the training set.
- **Generalization Issues**: A model that overfits to a small dataset will struggle to adapt to varied inputs in real-world applications, limiting its usefulness and robustness.
- **Limited Knowledge Transfer**: The essence of pretraining is to capture broad knowledge that can be transferred to specific tasks. Small datasets constrain the model's exposure to diverse language phenomena, resulting in limited transferability of learned representations to different tasks.
- **Task-Specific Challenges**: Inadequate pretraining may necessitate extensive fine-tuning for each new task, which is resource-intensive and may still not yield optimal performance due to the weak foundation established during pretraining.
- **Data Bias**: Small datasets are prone to biases, as they may not represent the full spectrum of language use across different domains, or topics. This bias can be ingrained in the pretrained model, leading to skewed or unfair predictions.

Small datasets pose a critical issue during the pretraining stage of transformers by limiting the model's ability to learn diverse, generalizable language patterns, increasing the risk of overfitting, reducing transferability, and introducing biases.

The next section explains the github API, the solution to gather a dataset with a rich representation and variability of information about software projects.

5.1 Github API Introduction

GitHub, a platform primarily known for hosting and managing software development projects, offers a powerful and comprehensive API (Application Programming Interface). The GitHub API, GitHub (2024), allows developers to interact programmatically with GitHub repositories, issues, pull requests, and other areas of the GitHub ecosystem. This capability is instrumental for automating workflows, integrating with other tools, and managing large-scale projects efficiently.

The GitHub API serves multiple purposes:

- **Data Retrieval**: Access and analyze data about repositories, users, and organizations to gain insights and make informed decisions.
- **Automation**: Automate repetitive tasks such as managing issues, pull requests, and releases, thereby reducing manual effort and human error.
- **Integration**: Seamlessly integrate GitHub with other tools and services such as CI/CD pipelines, project management tools, and communication platforms.
- **Customization**: Customize and extend the functionality of GitHub to meet specific project or organizational needs.

The GitHub REST API is the core interface for interacting with GitHub. It provides a wide array of endpoints for various operations:

- **Repositories**: Create, update, delete, and manage repositories.
- **Issues**: Create, manage, and close issues.
- **Pull Requests**: Create, review, and merge pull requests.
- **Commits**: Access commit history, compare commits, and manage branches.
- **Users**: Retrieve user information and manage user-specific settings.
- **Organizations**: Manage organization settings, members, and teams.

In addition to the REST API, GitHub offers a GraphQL API, which provides a more flexible and efficient way to query and manipulate data. With GraphQL, clients can request exactly the data they need, reducing the amount of redundant data transferred over the network.

5.2 The Best Open-source Projects

GitHub is a treasure trove of open-source projects across various domains and technologies. Identifying the best open-source projects on GitHub can be beneficial for developers seeking quality codebases, contributors looking to participate in established projects, or organizations aiming to integrate proven solutions. This section outlines methods to fetch the best open-source projects from GitHub using the GitHub API and other available resources.

Before fetching projects, it is essential to define what constitutes the ”best” open-source projects. Common criteria include:

- **Popularity**: Measured by the number of stars, forks, and watchers.
- **Activity**: Frequency of commits, issue resolution, and pull request merges.
- **Community**: Number of contributors and level of community engagement.
- **Documentation**: Quality and comprehensiveness of the project’s documentation.

- **Issues**: Number of open issues versus closed issues and the time taken to resolve issues.

The GitHub API provides endpoints to search for repositories and filter them based on various criteria. Below is a step-by-step guide to fetching the best open-source projects.

The search repositories endpoint can be used to find repositories based on specific criteria. The example below searches for repositories with the keyword ”machine learning” and sorts them by the number of stars, an example is detailed in 5.1.

```
curl -H "Authorization: token PERSONAL_ACCESS_TOKEN" \
"https://api.github.com/search/repositories
?q=machine+learning&sort=stars&order=desc"
```

Listing 5.1: Example Github API query.

The JSON response can be parsed to extract relevant details, example 5.2.

```
{
  "total_count": 12345,
  "items": [
    {
      "id": 123456,
      "name": "tensorflow",
      "full_name": "tensorflow/tensorflow",
      "html_url": "https://github.com/tensorflow/tensorflow",
      "description": "An open-source machine learning framework.",
      "stargazers_count": 150000,
      "forks_count": 80000,
      "open_issues_count": 1000,
      "updated_at": "2024-06-23T12:34:56Z"
    },
    ...
  ]
}
```

Listing 5.2: Example Github API response

5.3 Git Cloning

The main objective of this code is to clone a Git repository from a specified URL to a local directory. It ensures reliability by retrying the clone operation up to three times if it encounters any error. The code achieves this through an asynchronous function that manages environment variables and handles the cloning process, reporting success or failure and triggering retries if necessary, example **??**].

```
@backoff.on_exception(backoff.expo, asyncio.TimeoutError, max_tries=3)
async def clone_repo(repo_url, repo_id: str):
    repo_path = f'{os.getenv("REPO_CLONE_PATH", "/data/projects")}/{repo_id}'

    env = os.environ.copy()
    env.update({"GIT_LFS_SKIP_SMUDGE": "1", "GIT_ASKPASS": "echo"})

    process = await asyncio.create_subprocess_shell(
        f"git clone --depth 1 {repo_url} {repo_path}",
        env=env,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE,
    )

    _, stderr = await process.communicate()

    if process.returncode == 0:
        print(f"Successfully cloned {repo_url} into {repo_path}")
    else:
        print(f"Failed to clone {repo_url}: {stderr.decode().strip()}")
        raise asyncio.TimeoutError("Clone failed; will attempt to retry...")
```

Listing 5.3: Example git cloning in a subprocess

5.4 Project Tree Build

The eza command is a modern replacement for the ls command in Unix-based systems. It provides a more user-friendly and visually appealing way to list directory contents. Key features include color-coded file types, tree views of directories, filters with regex, integration with git ignore, icons for files, and better default options for displaying file sizes, permissions, and metadata. eza enhances the user experience by making it easier to understand and navigate file structures at a glance.

The code listed in Listing 3.4 is responsible for managing the build of project structures for software projects. It does this by using the eza command as a subprocess in the principal web service of the project. The script enables a safe mechanism to build the structure by creating a lock before attempting any updates to the repository, ensuring thread safety. Finally, the command is executed, have a look at code listing 5.4.

```
async def build_tree(commit: str, repo_id: str) -> str | None:
    repo_path = f'{os.getenv("REPO_CLONE_PATH", "/data/projects")}/{repo_id}'
    lock_path = f"{repo_path}/.lock"

    try:
        await __sleep_while_lock_exists(lock_path)

        with open(lock_path, "x") as _:
            print(f"Lock file created on repository {repo_id}")

        await __checkout(repo_path, commit)

        print(f"Running exa in {repo_path}")
        exa_process = await asyncio.create_subprocess_shell(
            "exa -TSBF --git-ignore -I 'regex(...)' | grep -vE '[0-9]{2}'",
            cwd=repo_path,
            stderr=asyncio.subprocess.PIPE,
            stdout=asyncio.subprocess.PIPE,
        )

        stdout, stderr = await exa_process.communicate()
        error_msg = stderr.decode()
        if exa_process.returncode != 0:
            if error_msg:
                print(f"Error running exa in {repo_path}: {error_msg}")
                raise ValueError(
                    f"Error running exa command, aborting. Error: {error_msg}"
                )
            print("exa command failed")

        return stdout.decode()

    except Exception as e:
        print(e)
        return None

    finally:
        if os.path.exists(lock_path):
            os.remove(lock_path)
            print(f"Lock file deleted on {repo_path}")
```

Listing 5.4: Example project tree build with a subprocess.

Figure 5.2 explains how data flows from its original form to the preprocessed state before being fed to the tokenizer. The following steps outline the process:

1. The `eza` command gathers the project structure.
2. Data is filtered to exclude files listed in the `.gitignore` file.
3. Data is further filtered using regex patterns to exclude common patterns such as `migrations`, `docs`, `translations`, and files with extensions like `*.png`.
4. Finally, data is filtered to avoid any file or directory containing a sequence of two or more numbers. This step is designed to exclude hashes and files that the AI model cannot understand due to a lack of contextual relevance.

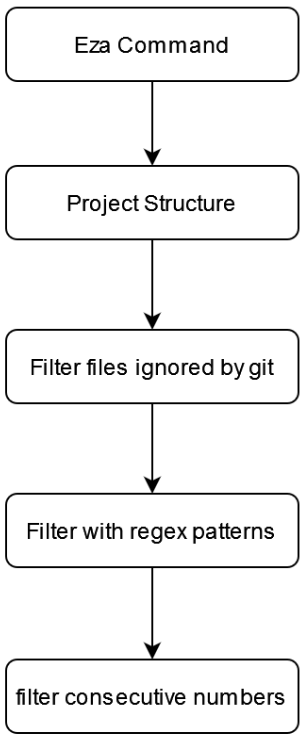

Eza Command
Project Structure
Filter files ignored by git
Filter with regex patterns
filter consecutive numbers

Figure 5.2: Project's tree data flow

6. Deployment Details

This chapter details the deployment strategy employed for the project. Several services were provisioned to ensure the system's functionality and accessibility. These services encompass:

- WebSite: The front-end interface for user interaction.
- WebService : The Web service hosting the API and reponsible for the system's functionality.
- Development Jupyter Lab: An interactive environment for development and experimentation.
- Internal Large Language Model (LLM): Local LLM for integration.
- Documentation: Model training guides, API documentation and code documentation.

A technical overview of the technologies leveraged in this process is presented to provide context on how a fully self-hosted deployment was achieved, without reliance on external cloud services. All services, including model training and model inference (both the trained Longformer model and local LLMs), were safely hosted by the intern. This required responsibility for not only software development but also hardware infrastructure design, network management (including the use of a static IP address and independent router), and the comprehensive security and maintenance of these elements.

6.1 Infrastructure Deployment and Hosting Process

This section outlines the comprehensive process undertaken to deploy and host the project's infrastructure, ensuring a secure, scalable, and high-performance environment.

1. **Infrastructure Setup and Validation:** The initial phase involved the meticulous validation and configuration of web and AI servers, network switches, and the primary router. Quality assurance measures were implemented to verify the integrity and stability of the hardware components.
2. **Router Configuration:** The *pfsense* router was configured with a static IP address, forwarding traffic to the ISP gateway. Network Address Translation (NAT) was implemented, incorporating port filtering and forwarding rules. Additionally,

an *OpenVPN*-based Virtual Private Network (VPN) was established to enable secure remote access.

3. **Router Security Enhancements:** The router was configured to support HTTPS 3 over QUIC using UDP on port 443. SSH access was also enabled for both the AI and web servers via the VPN, facilitating secure administrative access.

4. **Domain Management:** The project domain was registered, and DNS servers (*Cloudflare*) were updated to include domain and relevant subdomains redirections to the configured static IP.

5. **Nginx Reverse Proxy Implementation:** *Nginx* was deployed as the primary reverse proxy, enforcing TLS v1.2 or v1.3 and prioritizing HTTPS 3 for enhanced security and performance. Nginx also functioned as a load balancer, web server, and reverse proxy, implementing connection and bandwidth limits, filtering traffic based on IP addresses, and restricting WebSocket connections where unnecessary. Access was limited to authorized connections originating from the intern's or company VPNs.

6. **Automated Certificate Management:** Domain certificates were automatically validated and provisioned from *Let's Encrypt* through the integration of Nginx with *NixOS*. This streamlined process ensured the continuous availability of valid certificates and automated their renewall

7. **Jupyter Lab Service Hosting:** The Jupyter Lab service was securely exposed through the reverse proxy, enabling collaborative access for teams while maintaining stringent security protocols.

8. **Website and Documentation Hosting:** Nginx facilitated the hosting of static website and documentation files, ensuring scalability and production readiness. The primary domain was designated for the main website, while a subdomain ('docs') was allocated for documentation.

9. **API Service Proxying:** The web service API was proxied to a local IP address, incorporating TLS encryption and additional security measures. This configuration enabled horizontal scaling of the API service using the load balancer and permitted WebSocket connections on the 'apiv1' subdomain.

10. **Server Hardening and Service Configuration:** Servers were secured using stable builds with the latest security updates (*NixOS* 24.05), hardware stability was also validated. *Postgres* and *Redis* services were integrated into the production environment.

11. **Stress and Security Testing:** The infrastructure underwent comprehensive stress and security testing to assess its resilience and identify potential vulnerabilities. Continuous monitoring was implemented throughout the deployment and AI model training processes.

This systematic approach to infrastructure deployment and hosting ensured the project's successful launch, prioritizing security, scalability, and optimal performance.

6.2 Hosting System - A Technological View

This chapter briefly outlines the principal technologies involved in the deployment process, encompassing pfSense, Nix and NixOS, and NGINX as the primary technologies.

6.2.1 Introduction to PfSense

pfSense,Netgate (2024), is an open-source platform based on FreeBSD and is widely recognized for its reliability and extensive feature set. It is typically deployed in various settings ranging from small home networks to large corporate environments. This section evaluates pfSense as a primary network management solution.

As a router, pfSense offers advanced routing capabilities, support for multiple WAN connections, and load balancing. It handles complex routing protocols and network configurations with ease.

Key Routing Features

- Multiple WAN Support for enhanced redundancy and traffic management.
- Load Balancing to optimize bandwidth usage across various channels.
- Advanced Traffic Management with Quality of Service (QoS) rules.

The firewall capabilities of pfSense include stateful packet inspection, real-time traffic analysis, and deep packet inspection, providing secure network environments.

Key Firewall Features

- User-Friendly Web Interface for easy configuration of firewall rules and policies.
- Transparent Layer 2 Firewalling capabilities.
- Anti-Spoofing features to block potentially harmful traffic.

pfSense supports multiple VPN protocols such as IPsec, OpenVPN, and WireGuard, making it an excellent choice for secure remote access and site-to-site connections.

Deploying pfSense as a router, firewall, and VPN gateway provides an integrated solution that is scalable, robust, cost-effective and enterprise ready. This flexibility makes pfSense an excellent choice for organizations seeking to enhance their network infrastructure with a reliable, comprehensive tool.

6.2.2 DNS Servers

DNS servers, act as the internet's phonebook, translating human-readable domain names into machine-readable IP addresses. This allows web browser or more generally devices to locate and connect to the correct server hosting the website the user wants to visit.

6.2.3 Domain Certificate

Domain validation from Let's Encrypt is a crucial step in obtaining an SSL/TLS certificate, which encrypts communication between the website and its visitors. Let's Encrypt verifies the control of the domain by having the proven ownership through a challenge. This involves adding a file to the website's server. Once Let's Encrypt confirms the ownership of the domain, it issues a free SSL/TLS certificate.

6.2.4 System Management - Nix and Nixos an Introduction

In the landscape of Linux distributions and package management systems, NixOS presents a unique approach, aimed at improving configuration management, package isolation, and system reproducibility. NixOS is built around the Nix package manager, utilizing a purely functional deployment model. This section outlines the fundamental components of NixOS: NixOS itself, Nixpkgs, and Nix Shell..

NixOS,Foundation (2024), is a Linux distribution that employs a declarative model to manage system configurations and software packages. It leverages the Nix package manager, which tracks dependencies by building packages in isolation and ensuring that they do not interfere with each other. This approach eliminates common problems associated with dependency conflicts and inconsistent environments.

Key Features

- **Reproducibility:** NixOS configurations can be replicated across multiple machines effortlessly. This is because the entire system configuration, including the kernel, applications, system packages, and services, is built by the Nix package manager from a single configuration file (`configuration.nix`).
- **Rollbacks and Upgrades:** NixOS supports atomic upgrades and rollbacks. This means that system updates and configuration changes can be undone safely without affecting the stability of the system.
- **Isolation:** Applications and their dependencies are kept in separate directories, and different versions of a package can coexist on the same system without conflict.

NixOS is particularly popular in environments where system configuration and stability are critical, such as servers and development environments. It is favored by developers who require consistent and controlled environments.

Nixpkgs is the repository of package expressions used by the Nix package manager. It contains a comprehensive collection of definitions for building software packages and includes packages for applications, libraries, and system services.

- **Package definitions:** Each package in Nixpkgs has a corresponding Nix expression which describes how to build the package, including where to download the source, how to compile it, and the dependencies it requires.
- **Channels:** Nixpkgs supports channels, which are snapshots of Nixpkgs at a particular point in time, ensuring users have access to a consistent set of packages.

The breadth and organization of Nixpkgs make it a powerful resource for Nix and NixOS users, providing a stable base of packages that are easy to fetch, install, and manage through the Nix package manager.

Nix Shell is a tool provided by the Nix package manager designed to create reproducible development environments. It allows developers to enter a shell where specific dependencies are available, without permanently installing them on the system.

Features

- **Temporary environments:** Nix Shell can instantiate an environment for the duration of a session, ideal for testing and development without affecting the global system state.
- **Scriptable:** Developers can create `shell.nix` files that specify all dependencies required for a project, ensuring that any developer can replicate the environment.

Nix Shell is particularly useful for software development, where it is important to maintain project-specific dependencies that might vary from one project to another. It ensures that all contributors to a project work with the same setup, thus avoiding the ”works on my machine” syndrome, code listing 6.1 is provided as an example.

```
{ pkgs ? import <nixpkgs> {} }:
(pkgs.buildFHSUserEnv {
  name = "cuda-py311";
  targetPkgs = pkgs: with pkgs; [
    python311

    python311Packages.numpy
    python311Packages.torch
    python311Packages.transformers
    python311Packages.scipy
    # (...)

    openssl glibc libcxx gcc11
    git zip gitRepo gnupg autoconf
    curl procps gnumake cmake clang
    util-linux m4 gperf unzip

    cudatoolkit # Nvidia cudatoolkit

    linuxPackages.nvidia_x11
    libGLU libGL
    xorg.libXi xorg.libXmu freeglut
    xorg.libXext xorg.libX11 xorg.libXv xorg.libXrandr zlib
    ncurses5 stdenv.cc binutils
  ];
  multiPkgs = pkgs: with pkgs; [ zlib ];
  # runScript = "
  # fish
  # ";
  profile = ''
    export CUDA_PATH=${pkgs.cudatoolkit}
    export LD_LIBRARY_PATH=${pkgs.linuxPackages.nvidia_x11}/lib
    export EXTRA_LDFLAGS="-L/lib -L${pkgs.linuxPackages.nvidia_x11}/lib"
    export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:${pkgs.glibc}/lib";
    export EXTRA_CCFLAGS="-I/usr/include"
  '';
}).env
```

Listing 6.1: Example nix-shell configuration for AI projects.

NixOS, Nixpkgs, and Nix Shell form an ecosystem that offers significant advantages in terms of system configuration, package management, and development environment reproducibility. This model addresses common challenges in software environments, making it an attractive choice for developers and system administrators looking for robust, reliable, and reproducible computing environments.

6.2.5 Nginx an Introduction

A reverse proxy, Beyer et al. (2016), is a type of server that sits in front of web servers and forwards client (e.g., web browser) requests to those web servers. The reverse proxy provides a central point of control and can simplify the web infrastructure for scalability, security, and performance. Unlike a forward proxy, which protects clients, a reverse proxy is designed to protect and streamline access to servers.

1. **Load Balancing:** Distributes client requests across multiple servers to balance the load, ensuring no single server becomes overwhelmed, thus enhancing the performance and reliability of the web services.

2. **SSL Termination:** Handles incoming SSL connections, decrypting requests and passing unencrypted requests to the web servers, which reduces the SSL processing burden on the backend servers.

3. **Caching:** Temporarily stores frequently accessed content, reducing the need to repeatedly generate the same content by the backend servers, thereby improving response times and reducing load.

4. **Compression:** Compresses server responses before sending them to clients which helps in speeding up the transfer of information over the network.

5. **Security:** Provides an additional layer of defense, shielding backend servers from direct Internet exposure and various attacks.

Nginx, Nginx (2024), is a popular open-source software used for web serving, reverse proxying, caching, load balancing, and more. Here's how Nginx can be configured (nixos config) to act as a reverse proxy, web server, and load balancer:

```
services.nginx = {
    enable = true;
    package = pkgs.nginxQuic;

    recommendedProxySettings = true;
    recommendedGzipSettings = true;
    recommendedOptimisation = true;
    recommendedTlsSettings = true;

    logError = "stderr info";

    upstreams."backend_servers" = {
        servers = {
            "server1" = {
                address = "192.168.0.22:8082";
                backup = false;  # Primary Server
            };
            "server2" = {
                address = "192.168.0.23:8082";
                backup = false;  # Primary Server
            };
        };

        extraConfig = ''
            least_conn; # load balancing algorithm
        '';

    };

    virtualHosts = {
        "apiv1.domain.example" = {
            http3 = true;
            quic = true;
            forceSSL = true;
            enableACME = true;
            kTLS = true;

            locations."/" = {
                proxyWebsockets = true;
                proxyPass = "http://backend_servers";

                extraConfig = ''
                    allow 00.00.00.00;      # Vpn1 (hidden)
                    allow 192.168.192.0/24; # Vpn1 Network Tunel
                    allow 00.00.00.00;      # Vpn2 (Company) (hidden)

                    deny all;

                    proxy_set_header X-Real-IP $remote_addr;
                    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                    proxy_set_header X-Forwarded-Proto $scheme;
                '';
            };
        };
        "domain.example" = {
            # Similar for static files hosting
            # Acting as a web server for the website
            default = true;
            # (...)

            locations."/" = {
                alias = "/srv/project-cls/web/";
                tryFiles = "$uri /index.html";

                extraConfig = ''
                    index index.html
                    # (...)
                '';
            };
        };
    };
};
```

Listing 6.2: Example Nginx setup (nixos config) as a reverse proxy, load balancer and webServer.

The code listing 6.2 specifies example settings for an Nginx server managed through a declarative configuration tool, nix configuration.

Optimization and Security Settings

- Settings for proxy, compression, optimization, and TLS enhancements are activated to implement best practices (recommendedProxySettings, recommendedGzipSettings, recommendedOptimisation, recommendedTlsSettings).

Upstream Configuration

- **Servers**: Configures two primary servers, balancing load without redundancy.
- **Load Balancing**: Employs the least connections strategy (`least_conn`).

Virtual Hosts Configuration

- **Host** `apiv1.domain.example`:
  - Enables HTTP/3 usage, QUIC, enforced SSL, ACME, and kernel space TLS.
  - Manages access through IP whitelisting and comprehensive denial for other accesses.
  - Configures headers to preserve client IP and protocol integrity.
- **Host** `domain.example`:
  - Similar security and optimization configurations.
  - Serves static content efficiently, used for the website and docs build.

This example configuration, similar to the configuration for the project, strategically combines high-performance protocols with strict security measures to establish a robust, responsive web serving infrastructure, ready to manage extensive web traffic securely and efficiently.

7. Analysis of Results

This chapter presents a comprehensive analysis of the model's performance, encompassing training statistics, evaluation metrics, and the overall impact of the training process.

7.1 Training Statistics

The model underwent extensive training, utilizing a vast dataset of open-source projects. Key training statistics are summarized below:

- **Pretraining:**
  - Dataset: 60,000 projects (2.6 TiB) (600 MiB raw project structures)
  - Duration: 228 hours
  - Electricity Consumption: 146 Kw/h
- **Finetuning:**
  - Dataset: 3,000 projects (2,000 for training)
  - Duration: 12 hours
  - Electricity Consumption: 2 Kw/h
- **Annotation (using LLM):**
  - Duration: 146 hours
  - Electricity Consumption: 64 Kw/h
- **Research and Development:**
  - Duration: 320 hours
  - Electricity Consumption: 96.6 Kw/h
- **Hardware Resources:**
  - Computational Power: 35.6 TFLOPS (fp32), 71.2 TFLOPS (fp16)
  - Memory: 24 GB VRAM, 64 GB RAM

7.2 Base Model Evaluation

The base model, trained using masked language modeling, demonstrated a significant reduction in loss, from 12 to 0.7. This decrease indicates the model's capacity to learn the underlying patterns and structures of information effectively. While a specific evaluation dataset was not available, the low loss suggests that the model has acquired a strong contextual understanding and predictive capability without overfitting, overfitting would most likely be detrimental at this stage, so the pretraining did not go further in reducing the loss.

7.3 Finetuned Model Evaluation

The finetuned model achieved an accuracy of approximately 92.5% across all classes. This performance was validated on a dataset of 600 elements (91.9% accuracy) and tested on a different set of 600 elements (93.1% accuracy). While training accuracy reached 98%, further experimentation revealed that exceeding this threshold led to overfitting, with validation and test accuracy dropping to around 90%.

These results highlight the model's potential, particularly considering the diversity and complexity of the training data. It's noteworthy that the model was trained on a broad range of projects from GitHub, encompassing various languages and frameworks, without human supervision. Given the overfitting the training data size was not considered suitable.

Overall, the analysis indicates that the model's architecture and training methodology are sound.

8. Future Work and Considerations

This chapter outlines potential directions for future development of this project and reflects on the current state of implementation.

8.1 Annotations and Feedbacks from Engineers

The project currently facilitates a mechanism for engineers to annotate datasets and provide feedback on inference requests. This enables engineers (users) to efficiently annotate data in a structured manner, ensuring it is effectively captured by the system and used to enhance model accuracy. The supervision of data quality by qualified personnel is a critical factor for the continued success and value of this project in a production environment.

8.2 Additional Evaluation Metrics

On a dataset that was carefully annotated from engineers, the implementation of a confusion matrix derives substantial value for analysis of model performance. This was not implemented in the system due the lack of supervision in the dataset annotation, which means that the analysis of the matrix would not be as straighfoward.

In multi-label classification, where each data point can belong to multiple classes simultaneously, confusion matrices provide a comprehensive evaluation tool. They offer a detailed breakdown of prediction errors, highlighting not only the number of misclassifications per class but also the types of errors made (false positives and false negatives). This granular information is crucial for understanding model performance, identifying areas for improvement, and guiding the selection of appropriate evaluation metrics tailored to specific research objectives.

8.3 Distributed AI Systems

This model was trained on a single GPU. However, for production models where accuracy and training time are critical, this is not a viable solution.

This section explains how to distribute the training load across multiple GPUs or systems to improve performance and scalability.

Distributed training is a method used to scale deep learning models by splitting the computational workload across multiple hardware resources. This approach is crucial when training large models or using large datasets that do not fit into the memory of a single device. PyTorch offers several strategies to handle distributed training: Data Parallelism, Model Parallelism, and Hybrid approaches.

8.3.1 Data Parallelism

Data parallelism involves dividing the input data across multiple processing units, such as GPUs, and performing training simultaneously on each unit. Each processing unit operates a complete model replica. During the forward pass, each unit processes a subset of the data, generating corresponding model outputs and gradients. In the backward pass, gradients from all units are aggregated (typically via a method like All-Reduce) to update the model weights consistently across all units.

PyTorch implements data parallelism through its DataParallel and DistributedDataParallel (DDP) modules. The DataParallel module is simpler to use but less efficient compared to DDP as it does not scale as well across multiple nodes and can incur higher communication overhead due to the way it handles gradient aggregation.

DistributedDataParallel, on the other hand, is optimized for multi-node training. It reduces communication overhead by restricting gradient exchange to within-process or within-node, depending on the configuration. This method is highly recommended for multi-GPU or multi-node environments due to its efficiency and scalability.

8.3.2 Model Parallelism

Model parallelism involves splitting the model itself across different computational resources. This technique is useful when a single model does not fit into the memory of a single GPU or processor. In model parallelism, different parts of the network are located on different devices, and operations are synchronized across the devices as required.

Implementing model parallelism in PyTorch can be done manually by assigning different parts of a model to specific devices. For example, one might place the first few layers of a neural network on one GPU and the remaining layers on another. During training, data passes through the network across the devices, requiring synchronization points between the layers that are on different GPUs. This approach often involves handling device-to-device communication explicitly, which can be complex and may introduce significant overhead.

8.3.3 Hybrid Aproach

The hybrid approach combines elements of both data and model parallelism. In this approach, the model is partially divided across multiple devices (model parallelism), and each segment of the model processes different subsets of the data (data parallelism). This method can be advantageous when training extremely large models on large datasets and can help mitigate the limitations of each individual approach.

8.4 Rationale for Model Complexity

The overall model complexity aligns with the requirements of the current problem. A maximum sequence length of 4096 was employed, proving sufficient for classifying projects after pre-processing. However, for tasks involving complex or uncommon token relationships, a model with more parameters may be better suited. If the problem demands it, adjusting hyperparameters such as the number of attention heads, hidden layer dimensionality, or sequence length could be explored to integrate larger attention windows. In this project, model complexity was carefully considered and deemed appropriate for the task at hand.

8.5 Rationale for DataSet Size

The dataset size for the pretraining stage, encompassing approximately 60,000 projects, was deemed diverse and sufficient. This volume of information allowed the base model to grasp common patterns and relationships within the data.

However, the dataset used for finetuning presented challenges. It lacked proper supervision by engineers and was limited in size, comprising only 2,000 annotations. This quantity is insufficient for analyzing project structures that encompass a wide range of programming languages, frameworks, methodologies, and unconventional coding styles. Moreover, the annotated data was incomplete and lacked diversity. Therefore, the finetuning dataset is considered inadequate and requires increased volume, as well as thorough quality assurance, to effectively support model refinement.

9. Conclusion

This internship project successfully addressed a critical challenge in the area of Static Application Security Testing (SAST): the efficient and accurate identification of security vulnerabilities within the escalating complexity and volume of software data. By developing and implementing a modular, AI-driven sequence classification transformer from scratch, a significant step towards optimizing Checkmarx's query selection process was achieved.

Key Contributions

The key contributions of this project include:

- **Novel AI Solution for SAST:** The design and implementation of a versatile sequence classification model, trained and fine-tuned to effectively filter unnecessary queries within Checkmarx's SAST framework by classifying the type of software project.
- **Modular and Generic Framework:** The creation of a flexible solution adaptable to various domains and datasets, empowering future developers and researchers to leverage its capabilities.
- **Enhanced Efficiency and Accuracy:** The potential to significantly improve the speed and precision of Checkmarx's SAST tool by reducing the number of queries processed.

Personal Growth and Future Work

This internship has provided invaluable hands-on experience with cutting-edge AI techniques in the context of cybersecurity. The project not only deepened my understanding of language processing and transformer models but also highlights the transformative potential of AI in addressing real-world challenges in software security.

Future work specific to the project is detailed in chapter 8, looking ahead, several avenues for future work in the solution, SAST optimization, emerge:

- **Integration and Validation:** Fully integrating the sequence classification model into Checkmarx's SAST tool and conducting comprehensive validation on company codebases.

- **Model Enhancement:** Exploring more sophisticated transformer architectures and training strategies to further refine the model's performance.
- **Domain Adaptation:** Expanding the model's applicability to other areas of cybersecurity, such as malware detection and threat intelligence analysis.

Final Thoughts

Overall, this internship has been a rewarding journey. I am grateful for the opportunity to have contributed to Checkmarx's ongoing innovation in software security and look forward to witnessing the continued impact of AI in this field.

Acronyms & Abbreviations

AI	Artificial Intelligence. 5, 6, 8
CPU	Central Processing Unit. 9
GPU	Graphics Processing Unit. 9
TFLOPS	TeraFLOPS, a measure of a computer's speed and can be understood as a trillion floating-point operations per second. 9

Glossary

CodeBERT A variant of the BERT model trained specifically to understand and generate programming code. 5–7

DNS Domain Name System. The system that translates human-readable domain names into numerical IP addresses. 75

Fine-tuning The process of tuning a pre-trained model on a smaller, specific dataset to adapt it to a particular task or domain. 7, 8

HTTPS Hypertext Transfer Protocol Secure. An extension of HTTP that uses SSL/TLS to encrypt data. 74

NAT Network Address Translation. A method of remapping IP address space into another by modifying network address information in IP headers. 73

Nginx A web server used as a reverse proxy, load balancer, and HTTP cache. 78

NixOS A Linux distribution that uses a purely functional deployment model, managed by the Nix package manager. 76

NLP Natural Language Processing, a field of artificial intelligence focused on the interaction between computers and humans through natural language. 5, 8

pfSense An open-source firewall and router based on FreeBSD. 75

QUIC Quick UDP Internet Connections. A network protocol designed for encrypted, multiplexed, low-latency connections over UDP. 74

Sequence Classification A task in machine learning where a model is used to predict a category or class for a sequence of input data. 5–8

Tokenization The process of converting raw text into a format more suitable for model processing, typically by breaking text into words, subwords, or symbols. 7, 8

Transfer Learning Transfer Learning is a strategy in machine learning where knowledge gained while solving one problem is applied to a different but related problem. This technique involves taking a pre-trained model, which has already been developed and trained on a large, comprehensive dataset for a specific task, and fine-tuning it for a new task. This not only accelerates the development time by leveraging previously learned features and weights but also often improves performance when training data for the new task is limited. It is especially useful in domains where labeled data are scarce or expensive to obtain.. 5

Transformer A type of model architecture used for processing sequences, utilizing mechanisms like self-attention to process data in parallel and capture complex dependencies. 5, 6

VPN Virtual Private Network. A network that allows users to create a secure connection to another network over the Internet. 75

Bibliography

Beltagy, Iz, Matthew E Peters & Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* .

Beyer, Betsy, Chris Jones, Jennifer Petoff & Niall Richard Murphy. 2016. *Site reliability engineering: How google runs production systems*. O'Reilly Media.

Foundation, NixOS. 2024. Nixos manual. `https://nixos.org/manual/nixos/stable/`. Accessed on 20/06/2024.

GitHub. 2024. Github api documentation. Accessed: 2024-06-20. `https://docs.github.com/en/rest`.

Hugging Face. 2024. *Hugging face documentation*. `https://huggingface.co/docs`. Accessed: 2024-06-20.

Loshchilov, Ilya & Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* .

Netgate. 2024. pfsense documentation. `https://docs.netgate.com/`. Accessed on 20/06/2024.

Nginx, Inc. 2024. Nginx documentation. `https://nginx.org/en/docs/`. Accessed on 20/06/2024.

PyTorch. 2024. *Pytorch documentation*. `https://pytorch.org/docs/stable/index.html`. Accessed: 2024-06-20.

TensorFlow. 2024. *Tensorflow documentation*. `https://www.tensorflow.org/guide`. Accessed: 2024-06-30.

Xue, Linting, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts & Colin Raffel. 2022. Byt5: Towards a token-free future with pre-trained byte-to-byte models.

Yang, Jinbiao. 2024. Rethinking tokenization: Crafting better tokenizers for large language models.