HAW
HAMBURG

BACHELORTHESIS
Divyesh Joshi

Evaluation of Model Serving Frameworks for Machine Learning

FACULTY OF COMPUTER SCIENCE AND ENGINEERING
Department of Information and Electrical Engineering

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

HAMBURG UNIVERSITY
OF APPLIED SCIENCES

Hochschule für Angewandte
Wissenschaften Hamburg

Divyesh Joshi

Evaluation of Model Serving Frameworks for Machine Learning

Bachelor Thesis based on the examination and study regulations
for the Bachelor of Engineering degree programme
Bachelor of Science Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg
Supervising examiner: Prof. Dr. Kolja Eger
Second examiner: Prof. Dr. Wolfgang Renz

Day of delivery: 04. October 2024

Divyesh Joshi

Title of Thesis

Evaluation of Model Serving Frameworks for Machine Learning

Keywords

Machine Learning, Model Serving, TensorFlow Serving, Triton Inference Server, BentoML, FastAPI, Inference, Preprocessing, Postprocessing, REST API, Authentication, Latency, Performance, Cloud Deployment, Docker, Microsoft Azure

Abstract

This thesis presents an evaluation of model serving frameworks for machine learning, focusing on their performance, ease of deployment, and multi-model support in real-world production environments. The frameworks evaluated include TensorFlow Serving, Triton Inference Server, BentoML, TorchServe, and FastAPI. After a comprehensive theoretical analysis, TensorFlow Serving, Triton, and BentoML were selected for practical evaluation due to their compatibility with the project's requirements.
The final system integrates TensorFlow Serving with FastAPI to create a efficient machine learning model-serving platform. In this architecture, TensorFlow Serving handles inference while FastAPI is responsible for preprocessing, postprocessing, and implementing secure authentication using OAuth2. The system was tested under CPU-bound conditions using REST APIs to ensure broad compatibility.
Although TensorFlow Serving exhibited superior performance in terms of latency, testing on GPU-enabled hardware could potentially enhance performance across all frameworks, offering even greater improvements in inference speed and efficiency. Future work can focus on conducting more extensive testing, particularly on GPU-enabled systems.

Divyesh Joshi

Thema der Arbeit

Evaluation von Model-Serving Frameworks für maschinelles Lernen

Stichworte

Maschinelles Lernen, Model Serving, TensorFlow Serving, Triton Inference Server, BentoML, FastAPI, Inferenz, Preprocessing, Postprocessing, REST API, Authentifizierung, Latenz, Performance, Cloud- Deployment, Docker, Microsoft Azure

Kurzzusammenfassung

Diese Arbeit stellt eine Evaluierung von Model serving Frameworks für maschinelles Lernen vor und konzentriert sich dabei auf deren Leistung, einfache Bereitstellung und Multi-Modell-Unterstützung in realen Produktionsumgebungen. Zu den evaluierten Frameworks gehören TensorFlow Serving, Triton Inference Server, BentoML, TorchServe und FastAPI. Nach einer umfassenden theoretischen Analyse wurden TensorFlow Serving, Triton und BentoML aufgrund ihrer Kompatibilität mit den Anforderungen des Projekts für die praktische Evaluierung ausgewählt.

Das endgültige System integriert TensorFlow Serving mit FastAPI, um eine effiziente Plattform für maschinelles Lernen und Modellserving zu schaffen. In dieser Architektur übernimmt TensorFlow Serving die Inferenz, während FastAPI für das Preprocessing, Postprocessing und die Implementierung einer sicheren Authentifizierung mittels OAuth2 verantwortlich ist. Das System wurde unter CPU-gebundenen Bedingungen mit REST APIs getestet, um eine breite Kompatibilität zu gewährleisten.

Obwohl TensorFlow Serving eine überlegene Leistung in Bezug auf die Latenzzeit aufwies, könnte das Testen auf GPU-fähiger Hardware die Leistung aller Frameworks potenziell verbessern und noch größere Verbesserungen bei der Inferenzgeschwindigkeit und -effizienz bieten. Zukünftige Arbeiten können sich auf die Durchführung umfassenderer Tests konzentrieren, insbesondere auf GPU-fähigen Systemen.

Contents

List of Figures

List of Tables

Abbreviations and Symbols

ACI	Azure Container Instances
ACR	Azure Container Registry
API	Application Programming Interface
CNN	Convolutional Neural Networks
GRU	Gated Recurrent Unit
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JWT	JSON Web Tokens
ML	Machine Learning
LSTM	Long Short-Term Memory Networks
REST	Representational State Transfer
SSL	Secure Sockets Layer
TF	TensorFlow
TLS	Transport Layer Security
Triton	Triton Inference Server

1 Introduction

1.1 Motivation

Machine learning (ML) has become a growing area of research, attracting substantial attention from industry professionals. Deploying ML models is a critical step for their practical application in production environments. However, numerous industry reports and studies reveal that a large percentage of ML projects do not succeed in reaching deployment. This often leads to significant resource wastage, as companies invest considerable time and effort into projects that fail to be implemented effectively[22]. Model serving frameworks provide the essential infrastructure to deploy models. The motivation for this thesis arises from need to address these challenges by developing a secure and low-latency deployment system for ML models by evaluating model serving frameworks. The focus is on ensuring that the system provides controlled access to the model while maintaining data integrity and security.

1.2 Goals

The aim of this thesis is to create a design for a secure setup and implement it for deploying machine learning models. The system must ensure that only authenticated users can access the model's inference capabilities, while also guaranteeing secure data transmission between clients and the server. The system should be secure, easy to deploy, and adaptable to various environments, whether local or cloud-based. By achieving these objectives, the project aims to fill the gaps in existing model serving frameworks and provide a comprehensive, secure solution for machine learning model deployment.

1.3 Organization of Chapters

This thesis is organized into seven chapters.

Chapter 1 provides an introduction to the thesis and outlines the motivation behind building a secure system for machine learning model deployment.

Chapter 2 reviews the theoretical background, including machine learning model serving, data processing, security considerations, containerization and cloud deployment.

Chapter 3 outlines the requirements for the system, both functional and non-functional, and discusses the prioritization of these requirements using the MoSCoW method.

Chapter 4 presents the evaluation of various model-serving frameworks and discusses the design of the proposed system.

Chapter 5 explains the implementation details, covering how the system components were developed and integrated.

Chapter 6 selects the final setup based on performance evaluations, then conducts tests on the system.

Finally, Chapter 7 concludes the thesis, summarizing the findings and suggesting directions for future work.

2 Theory

In this section, the foundational concepts necessary for understanding the deployment of machine learning models are introduced. The discussion encompasses key topics such as machine learning model serving, data processing techniques, security measures, and cloud deployment strategies. These theoretical aspects are crucial for comprehending the challenges and decisions involved in designing a secure system for model inference.

The topics explored in this section provide the reader with a clear understanding of the underlying technologies and frameworks used in the deployment process. By addressing both the technical foundations, the theory section lays the groundwork for the implementation and evaluation phases that follow.

2.1 Machine Learning Model Serving

2.1.1 Overview of Model Serving

Machine learning has become a fundamental component of modern technology. As models become increasingly sophisticated, the challenge has evolved from merely developing accurate models to also ensuring their efficient deployment.

Model serving refers to deploying ML models in production environments, enabling them to be accessed and used as callable services over a network. In simple terms, it involves making a trained ML model available for real-world applications via a REST or gRPC API. Specifically, model serving includes setting up the infrastructure or system needed to host the models and handle network requests, allowing the model to deliver predictions in real time[24].

2.1.2 Model Serving Frameworks

Deploying ML models requires stable frameworks that can handle the complexities of production environments. Several model serving frameworks have emerged, each with unique features and capabilities.

Tensorflow Serving

TensorFlow Serving is an open-source, high-performance model serving system designed to deploy ML models in production environments. It supports serving multiple models or multiple versions of the same model simultaneously. It offers seamless integration with TensorFlow models. Key features include model versioning, efficient resource management, and support for both gRPC and HTTP endpoints, allowing flexible integration with various client applications and systems[77] .

TorchServe

TorchServe is an open source model serving framework developed, designed to efficiently deploy and serve PyTorch models in production. It enables users to easily host, manage, and scale PyTorch models with features like multi model serving, automatic batching, metrics for monitoring, and REST APIs for model inference. TorchServe supports a range of ML workflows, including model versioning, customizable inference logic, logging, and scaling for both cloud and on-premise environments. It also integrates with popular MLOps tools, making it easier to automate and optimize the model serving process[56].

Triton Inference Server

Triton Inference Server is developed by NVIDIA. It is an open source software platform that simplifies the deployment of AI models in production environments. Triton Inference Server allows teams to deploy AI models from a variety of deep learning and machine learning frameworks, such as TensorRT, TensorFlow, PyTorch, ONNX, OpenVINO, Python, RAPIDS FIL, and others. It supports inference across different environments, including cloud, data centers, edge, and embedded devices, and can run on NVIDIA

GPUs, x86 and ARM CPUs, as well as AWS Inferentia. Triton provides optimized performance for various query types, such as real-time, batch processing, ensemble models, and audio/video streaming [36].

BentoML

BentoML is a Python library designed for the creation of online serving systems that are specifically tailored for the implementation of artificial intelligence applications and model inference. It supports a variety of machine learning frameworks, including TensorFlow and PyTorch, allowing the definition of multiple services tailored to specific tasks, such as data preprocessing or model inference. Each service is fully customizable, allowing the user to manage its input and output logic as needed[9].

FastAPI

FastAPI is a modern and high-performance web framework designed for building APIs with Python. It is known for its speed, it is the fastest python framework available. FastAPI speeds up development, reducing coding time and bugs[59].

While FastAPI is not a dedicated model-serving framework, it is highly adaptable and can be integrated with machine learning tools to handle tasks such as pre-processing, post-processing, and managing inference requests[62].

2.2 Machine Learning Models

In this thesis, two pre-trained machine learning models are deployed for energy forecasting: a CNN-LSTM model and a GRU model. Both models are designed for time-series analysis and are used to predict future energy consumption based on historical data.

Both the CNN-LSTM and GRU models process time-series data with the same input and output structure:

- **Input**: The models take a time-series input of shape (None, 672, 7)
- **Output**: Both models generate a prediction vector of size 8, which represents the predicted values for the target variables related to energy consumption.

2.3 Data Processing

The process of data preprocessing involves the evaluation, filtering, manipulation, and encoding of data in a manner that enables a machine learning algorithm to comprehend and utilize the resulting output[35]. It is important to ensure that data is processed before and after the inference stage in order to guarantee that the models receive the expected input and that the output is readable by the user.

2.3.1 Input Data

The input data consists of two columns: Date-Time and Power Consumption. The Date-Time column records the timestamps at 15-minute intervals, while the Power Consumption column reflects the corresponding energy usage in kilowatts (kW) for each interval.

2.3.2 Preprocessing

The preprocessing of data for the purposes of machine learning (ML) refers to the preparation and transformation of raw data into a format that is suitable for the training of ML models. The techniques involved in data preprocessing include the cleaning and handling of missing values, the removal of outliers, the scaling of features, the encoding of categorical variables and the splitting of the data into two distinct sets, namely the training set and the testing set[72]. Key tasks in preprocessing include adding time-based features, normalization, and reshaping the data to meet the requirements of the model.

- **Adding Time-Based Features**: Time-based features are introduced using sinusoidal and cosinusoidal transformations to capture cyclical patterns such as hour, day, week, month, and year. These transformations help the model better understand and detect recurring trends in the data, improving its ability to forecast energy consumption over time[64].
- **Normalization**: Normalization in data preprocessing refers to the technique of scaling and standardizing the values of features within a data set. The primary objective of normalization is to align all feature values within a comparable range while maintaining the distinct ranges of values. This is crucial because numerous

machine learning algorithms demonstrate enhanced performance or faster convergence when the input features are on a comparable scale and exhibit a similar distribution.[72]. In this project, the data is normalized using the mean and standard deviation values for each feature, a common approach to ensure that all input features are on the same scale.

- **Data Reshaping**: Data reshaping is the process of changing the layout or structure of data from one form to another, in order to better suit the needs of an analysis or downstream processing. This can involve rearranging rows and columns, changing the data type, or converting between wide and long formats[12]. The data is organized into three dimensions: batch size, time steps, and features, allowing the model to capture patterns across multiple time points.

Through preprocessing, the raw input data is transformed into a structured, normalized format that the model can effectively use for accurate prediction.

2.3.3 Postprocessing

Postprocessing refers to the steps taken after the model generates predictions, which typically involve transforming the model's output into a usable format.

Since the model outputs normalized values, postprocessing involves denormalization to bring the predictions back to their original scale. Denormalization is accomplished by applying the inverse of the normalization process, multiplying the predictions by the standard deviation and adding the mean back to the results. This step makes sure that the predictions are in the correct format, making them interpretable for practical use.

2.3.4 Common Tools

Several libraries and frameworks are commonly used to facilitate preprocessing and postprocessing in machine learning workflows.

- **Pandas**: Pandas is a library in the Python programming language that provides data structures designed to facilitate the processing of data in a straightforward and intuitive manner.[74].

- **NumPy**: NumPy provides powerful numerical operations that are critical for normalization, reshaping, and array-based transformations[13].
- **Custom Python Code**: For specific use cases, custom implementations are often written to handle unique data processing requirements.

By utilizing these tools, the project ensures efficient and reliable data handling from preprocessing to postprocessing.

2.4 Security

In machine learning model deployment, ensuring the security of both the models and the data they process is crucial, particularly in production environments. This project implements a range of security measures, including OAuth2 for authentication, JWT for access control, and TLS for encrypted communication. These mechanisms together form a reliable setup to safeguard model inference APIs from unauthorized access and data breaches.

2.4.1 OAuth2 and JWT for Authentication and Access Control

OAuth2 is the industry standard protocol for authorization, commonly used to secure APIs and web applications[41]. The framework allows users to authenticate by obtaining access tokens. These are the credentials that are employed in order to gain access to protected resources. An access token is a string that represents an authorization issued to the client. The access token allows access to specific resources for a limited scope and duration. These tokens are granted by the resource owner and enforced by both the resource server and authorization server[23]. In the context of ML model deployment, implementing OAuth2 ensures that only authenticated users can interact with the model inference API, thereby securing access and preventing unauthorized use of sensitive data.

Bearer Tokens are the most commonly used type of access token in the OAuth2. A Bearer Token is essentially an opaque string that does not carry any intrinsic meaning to the clients using it. Different servers may issue tokens in various formats, some may use a simple hexadecimal string, while others might opt for a more structured format, such as JWTs[42].

JWTs are a specific type of Bearer Token that encodes claims in a JSON format, which is then signed for security. When used as an OAuth 2.0 Bearer Token, a JWT allows all relevant data, such as the user's identity and token expiration, to be included directly within the token itself. This eliminates the need for the server to store token information in a database, making JWTs an efficient and stateless method of managing access tokens in distributed systems[43].

2.4.2 TLS for Secure Communication

The most prevalent protocols for encryption in network communications are Secure Sockets Layer (SSL) and its successor, Transport Layer Security (TLS). Both facilitates the secure exchange of data by encrypting information transmitted between servers, applications, users, and systems. These protocols ensure the authentication of both parties in a network connection, which enables the transmission of data in a secure manner.[68].

Hypertext Transfer Protocol (HTTP) is a communication protocol that defines the rules for client-server interactions across networks. SSL and TLS encryption are used in Hypertext Transfer Protocol Secure (HTTPS) to enhance security for otherwise insecure HTTP connections. Prior to connecting to a website, a browser uses TLS to verify the site's SSL/TLS certificate, confirming that the server complies with current security standards.[68].

In this system, the TLS is utilized to encrypt data during transmission, thereby ensuring the secure protection of both model inputs and outputs against unauthorized access.

2.5 Containerization and Cloud Deployment

This section outlines the containerization strategy using Docker, service orchestration through Docker Compose, and cloud deployment leveraging Azure cloud services.

2.5.1 Containerization

Containerization represents a software deployment process whereby an application's code is bundled with all the requisite files and libraries, enabling its operation on any given infrastructure. In the past, the installation of an application on a computer required the user to install the version of the software that corresponded to the operating system of the machine in question. For instance, the Windows version of a software package would need to be installed on a Windows machine. However, the use of containerization allows the creation of a single software package, or container, that can be executed on a variety of devices and operating systems[69]. Unlike traditional virtualization, containers share the host system's kernel while isolating the application, making containers lightweight and more resource efficient[17].

In this project, Docker is used for containerizing the machine learning model and related services. Docker allows the creation of isolated environments where the model can be packaged along with its dependencies, including the operating system, libraries, and runtime. This guarantees consistency in both development and production environments, eliminating issues related to configuration mismatches. Docker allows containers to be easily deployed on local machines or cloud platforms [16].

2.5.2 Orchestrating Multiple Services

While individual containers provide isolation, real-world applications often consist of multiple services that need to communicate and work together. For example, a machine learning deployment may involve a model inference server, an API gateway, and a database. Docker Compose is used in this project to orchestrate these multiple services, allowing them to be defined and managed collectively. Docker Compose simplifies multi-container orchestration by using a single YAML file to define all the services, their networks, volumes, and dependencies. Additionally, it provides networking capabilities, ensuring that the services can communicate securely over internal Docker networks without requiring external configurations [14]. For instance, the ML model's inference server is deployed alongside an API server, where each service is defined as a container in the Docker Compose file. Docker Compose handles the service lifecycle, starting, stopping, and scaling containers as needed.

The advantage of using Docker Compose is its ease of use in development and production environments. Developers can bring up the entire system with a single command, ensuring that all services are properly configured and started in the correct sequence[15]. This orchestration is crucial for ensuring that the ML model, API, and any supporting services work in harmony without manual intervention.

2.5.3 Cloud Deployment

Once the system is containerized and orchestrated, it needs to be deployed in a flexible and reliable environment. Cloud deployment offers significant benefits, such as the ability to adjust resources as needed, high availability, and reduced infrastructure management. In this project, Microsoft Azure is chosen as the cloud platform, utilizing its services for container management and deployment.

- **Azure Container Registry (ACR)**: ACR is a managed service that allows users to store and manage Docker container images. ACR integrates seamlessly with other Azure services and supports the automated build and deployment of containerized applications. By pushing the Docker images to ACR, the images can be securely stored and retrieved during the deployment process, ensuring that the correct version of the application is always available for deployment[30].
- **Azure Container Instances (ACI)**: ACI provides a serverless environment for running Docker containers in the cloud. ACI allows containers to be deployed without the need for managing underlying infrastructure, simplifying the deployment and scaling processes. ACI supports the rapid scaling of containers based on demand, ensuring that the system can handle fluctuating workloads efficiently[31].

Additionally, ACI integrates with ACR, enabling seamless container image retrieval and deployment[32]. This integration allows the latest versions of the model and services to be deployed directly from the container registry without manual intervention.

3 Requirements

This chapter presents the function and non functional requirements that the system must fulfil in order to achieve the desired functionality and performance.

3.1 Functional Requirements

1. **FR1: TensorFlow Model Deployment Capability**
   The system must support the deployment of TensorFlow models. It should be capable of loading and serving TensorFlow models.
2. **FR2: Data Preprocessing**
   The system should support preprocessing of data before sending it to the model for inference. This includes tasks like normalization, transformation, and validation of input data to ensure it is ready for inference.
3. **FR3: Data Postprocessing**
   The system should support postprocessing of data after receiving results from the model. This involves formatting the inference results before sending them to the client for readability.
4. **FR4: Secure Authentication**
   The system must provide secure authentication. Clients must authenticate to access the inference API, ensuring only authorized users can interact with the system.
5. **FR5: API Support for Model Inference**
   The system must expose an API (e.g., REST or gRPC) that allows clients to send inference requests and receive model predictions.
6. **FR6: Local Deployment with Containerization**
   The system must support local deployment using containerization technologies such as Docker, to enable easy deployment and management in local environments.

7. **FR7: Compatibility with Cloud Providers**
   The system should integrate seamlessly with cloud services (e.g., Azure, AWS, or Google Cloud).

8. **FR8: Model Management**
   The system should be able to deploy and manage multiple TensorFlow models concurrently.

3.2 Non-Functional Requirements

1. **NFR1: Low Latency for Request and Response**
   The system should have a low latency, with a response time of less than 3 seconds. The majority of prevalent HTTP libraries, including Python's Requests, lack a default timeout mechanism[50]. Consequently, requests may remain unresolved indefinitely unless a specific timeout value is explicitly defined[28]. The default timeout period for OkHttp is 10 seconds, which may prove to be excessive for performance-critical applications[2]. In contrast, the HTTPX employs a default timeout period of 5 seconds[25]. Therefore, selecting a 3 second timeout represents a strategic choice, as it is shorter than the default for HTTPX, thereby ensuring faster response times and avoid a timeout by the client.

2. **NFR2: Secure Communication**
   The system must ensure secure communication between clients and servers. This includes encryption of data in transit, through HTTPS to protect against unauthorized access or tampering.

3. **NFR3: Framework Popularity and Support**
   The system should be built using popular and widely adopted frameworks with strong community support. This ensures that the system benefits from continuous updates, security patches, and long term support, making it future proof.

3.3 MoSCoW Priority Classification

The MoSCoW method is used to prioritize the system's requirements based on their importance and necessity. The classification is divided into four categories: Must Have, Should Have, Could Have, and Won't Have. Below is the arrangement of the priority classification for both functional and non-functional requirements[11].

3.3.1 Must Have

These features are critical requirements that the project cannot be completed without them. If these are not fulfilled, the project is considered a failure.

- **FR1: TensorFlow Model Deployment Capability**
- **FR2: Data Preprocessing**
- **FR3: Data Postprocessing**
- **FR4: Secure Authentication**
- **FR5: API Support for Model Inference**
- **FR6: Local Deployment with Containerization**
- **NFR1: Low Latency for Request and Response**
- **NFR2: Secure Communication**

3.3.2 Should Have

These features are important but not critical features of a project, and these are high-priority items that are not as time-sensitive as the Must-haves.

- **FR7: Compatibility with Cloud Providers**
- **FR8: Model Management**

3.3.3 Could Have

Desirable features that can be included if time and resources permit, but are not essential for the project's success:

- **NFR3: Framework Popularity and Support**

3.3.4 Summary of MoSCoW Prioritization

- **Must Have:** FR1, FR2, FR3, FR4, FR5, FR6, NFR1, NFR2
- **Should Have:** FR7, FR8
- **Could Have:** NFR3
- **Won't Have:** None explicitly defined for now.

4 Evaluation of Frameworks and Design

The objective of the design section of this thesis is to present a systematic approach to the evaluation and design of a stable and secure system for the deployment of machine learning models. This section commences with an evaluation of most common model serving runtime frameworks, specifically TensorFlow Serving, Triton Inference Server, TorchServe, BentoML, and FastAPI. The evaluation focuses on these open-source runtime frameworks due to their widespread usage[34]. The evaluation process allows for the identification of the most suitable candidate frameworks for further consideration.

Once the candidate frameworks have been selected, their individual designs are discussed in detail. Each design is tailored to address the specific strengths and limitations of the selected frameworks, with a focus on building a secure and efficient environment for machine learning inference, incorporating authentication, HTTPS encryption, ensuring the system has low response time to enhance performance. This step-by-step approach is essential to achieving the project's goal of providing a secure and production-ready deployment system.

4.1 Criteria for Selecting a Framework

In order to select the most appropriate frameworks for the deployment of TensorFlow models, a set of key criteria was established to guide the decision-making process. The criteria were derived from the specific requirements outlined in Chapter 3, which include factors critical to the successful deployment and operation of TensorFlow models in a production environment.

The following criteria were identified as essential for the evaluation of potential frameworks:

- **Tensorflow Model Deployment Capability**: The capability of a framework to efficiently deploy TensorFlow models is a fundamental criterion for this project, given the widespread use of TensorFlow in machine learning tasks. A suitable framework must support TensorFlow model.
- **Data Processing Capabilities**: Effective data processing, both before and after inference, is critical to ensuring the accuracy and usability of ML models. A framework must support comprehensive data preprocessing such as normalization, feature scaling, and transformation before the data is passed to the model. Similarly, postprocessing capabilities, including denormalization and formatting of output data, are necessary to ensure the results are interpretable.
- **Security Measures**: Given the sensitivity of data in ML applications, robust security mechanisms are critical. A suitable framework should provide TLS/SSL encryption to ensure secure communication between clients and servers. Additionally, the framework should offer built-in support for authentication mechanisms, such as OAuth2 and JWT, to control access to the model inference API and ensure that only authorized users can interact with the deployed models.
- **Local Deployment Capabilities**: The ability to deploy models on a local server is essential for organizations that prefer to maintain their infrastructure in-house or need to run models in environments with restricted or no access to cloud services.
- **Cloud Deployment Capabilities**: A framework should support seamless integration with major cloud platforms, enabling the system to take advantage of cloud services.
- **Performance**: The framework must exhibit low-latency performance, ensuring that predictions are returned quickly.
- **Model Management**: The framework must have the capability to handle the deployment and management of multiple TensorFlow models at the same time. This includes the ability to manage various models concurrently in a production environment, ensuring that different models can be served.
- **Ease of Deployment**: The level of ease or difficulty in setting up and deploying models.

- **Popularity**: The framework's popularity is an important consideration, as widely adopted solutions tend to have strong community support, frequent updates, and extensive resources. A popular framework is likely to be more future proof.

4.2 Evaluating frameworks against the criteria

4.2.1 TensorFlow Model Deployment Capability

- **TensorFlow Serving**: TensorFlow Serving is optimised for deploying TensorFlow models. It provides a streamlined process for serving models. It is tightly integrated with the TensorFlow ecosystem, enabling easy deployment of SavedModel formats[76].
- **TorchServe**: TorchServe does not support TensorFlow models. It is specifically designed to serve PyTorch models.
- **Triton Inference Server**: Triton Inference Server supports TensorFlow models. It can handle various TensorFlow formats, including TensorFlow 1.x and 2.x, TensorFlow SavedModel, and TensorFlow GraphDef[61].
- **BentoML**: BentoML is equipped with the capability to support TensorFlow models. The BentoML local Model Store can be utilised for the administration of these models, as well as for the construction of applications based on them[9].
- **FastAPI**: FastAPI is not a dedicated model serving framework but can be used to deploy TensorFlow models by wrapping them in a REST API[62].

4.2.2 Data Processing Capabilities

- **TensorFlow Serving**: TensorFlow Serving does not have native support for complex pre- and post-processing operations. It expects inputs to already be in the format required by the model. Simple pre- and post-processing steps can be included as part of the TensorFlow model graph using TensorFlow operations. More complex data processing is typically performed in the client application[26].

- **TorchServe**: TorchServe does not have complex pre-processing built in. It expects inputs to be in the format required by the model. Pre- and post-processing steps can be implemented in a custom handler class by overriding the preprocess. function[73].
- **Triton Inference Server**: Triton lacks the capacity for complex pre-processing operations; however, it provides the option of custom backends for the implementation of custom pre- and post-processing. The creation of custom C++ backends that perform pre- and post-processing prior to inference is a possibility[37].
- **BentoML**: BentoML facilitates the specification of customised services for specific tasks, including data pre- and post- processing. The implementation of pre-processing logic within a service class allows for the handling of operations such as data normalisation, transformation and feature extraction prior to the transfer of data to the model for inference[9].
- **FastAPI**: FastAPI provides a flexible environment in which custom pre-processing and post-processing logic can be implemented within API endpoints[71].

4.2.3 Security Measures

- **TensorFlow Serving**: TensorFlow Serving supports SSL/TLS encryption, ensuring secure communication between clients and servers[44]. However, it does not provide built-in authentication mechanisms, meaning authentication must be handled externally.
- **TorchServe**: TorchServe provides SSL/TLS encryption for secure communication[51], but like TensorFlow Serving, it does not offer native authentication mechanisms.
- **Triton Inference Server**: The Triton Inference Server supports SSL/TLS encryption, which secures communication between clients and the server, enhancing overall inference request security[38]. However, Triton does not include built-in authentication or authorization mechanisms.
- **BentoML**: BentoML supports SSL/TLS encryption for secure communication within its model serving framework[7], but does not provide native authentication mechanisms.

- **FastAPI**: FastAPI supports SSL/TLS encryption for secure communication[29] and also offers a comprehensive set of features for authentication, including support for OAuth2 and JWT, making it an optimal choice for developing secure APIs[21].

4.2.4 Local Deployment Capabilities

All of the selected frameworks support local deployment through Docker containers. Docker provides a consistent and isolated environment, making it easy to deploy models on any local setup without requiring extensive manual configuration or installation. By containerizing the entire model serving infrastructure, including dependencies and runtime environments, developers can ensure that the model serving setup behaves the same across different local machines.

- **Tensorflow Serving**: Tensorflow Serving can be deployed locally using Docker or directly on a machine[77].
- **TorchServe**: TorchServe can be run locally via Docker or directly on the system[55].
- **Triton Inference Server**: Triton Inference Server can be run locally via Docker[40].
- **BentoML**: BentoML can be run locally via Docker or directly on the system[6].
- **FastAPI**: FastAPI can serve ML models locally[19].

4.2.5 Cloud Deployment Capabilities

Docker containers offer significant flexibility for deploying model-serving frameworks in the cloud. Since all major cloud providers support Docker, the same containerized environments used for local development can easily be deployed on any cloud platform.

- **Tensorflow Serving**: Tensorflow Serving it is suitable for cloud deployment, particularly when integrated with Docker containers, which allows it to be run on various cloud platforms[78].
- **TorchServe**: TorchServe can be deployed to a cloud computing environment of the user's choosing[57].
- **Triton Inference Server**: The Triton Inference Server is capable of being deployed to a cloud computing environment selected by the user[57].

- **BentoML**: BentoML is capable of being deployed in a cloud computing environment selected by the user, or alternatively, on Bentocloud[57].
- **FastAPI**: The deployment of a FastAPI application may be accomplished through the utilisation of a broad range of cloud providers[18].

4.2.6 Performance

- **Tensorflow Serving**: TensorFlow Serving is optimized for serving machine learning models in production, particularly TensorFlow models. It provides fast model deployment by handling multiple versions and enabling easy updates without downtime. Its C++ implementation offers low latency and high performance. The architecture is highly modular, which supports advanced features like model batching, dynamic model loading, and GPU utilization.[60].
- **TorchServe**: Similar to TensorFlow Serving, TorchServe is optimized for serving PyTorch models in production[56].
- **Triton Inference Server**: Triton Inference Server is designed for high performance model serving across multiple frameworks and is optimised for NVIDIA GPUs[34].
- **BentoML**: BentoML is written in Python, which is not as optimal for performance as other frameworks such as Tensorflow Serving or TorchServe[34].
- **FastAPI**: Although FastAPI is capable of perform ML model inference, it is not as highly performant as frameworks that have been specifically designed for the purpose of serving ML models.[8].

4.2.7 Model Management

- **TensorFlow Serving**: TensorFlow Serving is capable of serving multiple models or versions concurrently. It offers versioning capabilities, enabling the deployment of new model versions without requiring modifications to the client code[77].
- **TorchServe**: TorchServe provides model versioning and management through the use of configuration files[54].

- **Triton Inference Server**: Triton Inference Server provides flexible and dynamic handling of models in production environments with strong model management capabilities[39].
- **BentoML**: BentoML facilitates the deployment and management of multiple models, including different versions, within a single servic[5].
- **FastAPI**: FastAPI does not provide any built-in model versioning features. There is no native support for managing multiple model versions, and therefore versioning would need to be implemented manually.

4.2.8 Ease of Deployment

- **TensorFlow Serving**: TensorFlow Serving allows for easy deployment by pulling a Docker image and mounting the model directory. The process involves minimal configuration, making it very easy to set up and ready to serve predictions via REST or gRPC endpoints[33].
- **TorchServe**: The deployment process involves installing TorchServe, archiving the model into a '.mar' file, and then starting the server to serve the model[52].
- **Triton Inference Server**: The deployment involves pulling the Docker image, providing a detailed model configuration file, and setting up a model repository. This process can be complex, especially for beginners[66].
- **BentoML**: Deploying with Bentoml requires BentoML to be installed in Python, then users need to define a service that wraps their model in a Python script. This requires some coding and can be complex for beginners[4].
- **FastAPI**: The deployment process entails the definition of application programming interface (API) endpoints for model inference, a process that can become complex and time-consuming. It is not a straightforward undertaking[62].

4.2.9 Popularity

- **TensorFlow Serving**: TensorFlow Serving is the most popular of the frameworks under consideration, with a total of 6.2k stars and 2.2k forks on GitHub[77]. Furthermore, the software has been downloaded on over 9 million occasions on PyPI over the past month, which is indicative of its extensive adoption and the strength of the community supporting it. The considerable user base and comprehensive resources make TensorFlow Serving an exceptionally reliable and future-proof choice for production environments[47].

- **TorchServe**: TorchServe has moderate popularity, especially within the PyTorch community. It has gained 4.1k stars and 835 forks on GitHub[53]. In terms of PyPI downloads, TorchServe recorded around 54k downloads last month[48].
- **Triton Inference Server**: Triton Inference Server shows strong popularity, especially for GPU-based deployments. It has over 8k stars and 1.4k forks on GitHub[65], reflecting significant interest in the high-performance machine learning space. With around 1.35 million downloads on PyPI last month, Triton remains a popular choice for enterprises and researchers looking for optimized performance[49].
- **BentoML**: BentoML is a framework that is experiencing a period of growth, with 6.9k stars and 775 forks on GitHub[4]. The popularity of the framework is increasing, with approximately 108k downloads on PyPI over the past month. This indicates that, despite its relative infancy, it is rapidly gaining traction among developers who value flexibility and ease of use in the construction and deployment of machine learning services[45].
- **FastAPI**: FastAPI enjoys immense popularity due to its nature as a general-purpose web framework. It has over 75k stars on GitHub and 6.4k forks[20]. With over 41 million downloads on PyPI last month, making it one of the most popular Python web frameworks. While it isn't specifically built for model serving, its popularity, large community, and extensive resources make it a flexible option for integrating model inference into web applications[46].

Table 4.1: Comparison of Model Serving Frameworks

Feature/Capability	TF Serving	TorchServe	Triton	BentoML	FastAPI
Model Deployment Capability	+++	×	+++	+++	+++
Data Processing Capabilities	+	+	+	+++	+++
Security Measures	+	+	+	+	+++
Local Deployment Capabilities	+++	+++	+++	+++	+++
Cloud Deployment Capabilities	+++	+++	+++	+++	+++
Performance	+++	+++	+++	+	×
Model Management	+++	+++	+++	+++	×
Ease of Deployment	++	++	×	++	×
Popularity	+++	+	++	+	+++
Total Score (Points)	**22**	**17**	**19**	**20**	**18**

4.3 Candidate Framework Selection and Justification

Based on the detailed comparison of model serving frameworks(see Table 4.1), several key decisions were made to narrow down the candidates for evaluation:

4.3.1 Exclusion of TorchServe and FastAPI

Although TorchServe is a competent serving framework for PyTorch models, it lacks the requisite support for TensorFlow models, which are central to this project's requirements. As TensorFlow models must be deployed, TorchServe was excluded from further consideration due to its inability to serve from the TensorFlow models(refer to Table 4.1 for TorchServe's limitations in model deployment capabilities).

Another reason to exclude TorchServe is that the framework is comparable to TensorFlow Serving in terms of functionality. As both frameworks offer similar capabilities, there is no advantage in transitioning to PyTorch models for the current work, which already relies heavily on TensorFlow models. This approach ensures that compatibility and consistency within the existing data science workflows.

Similarly, FastAPI, was excluded from being used as a standalone model serving solution. As detailed in Table 4.1, FastAPI is not a dedicated model serving framework, while it offers excellent capabilities for handling preprocessing, postprocessing, and authentication, it lacks the direct inference capabilities provided by TensorFlow Serving, Triton, or BentoML.

4.3.2 Selected Candidates for Evaluation

The remaining candidates, namely TensorFlow Serving, Triton Inference Server, and BentoML, were selected for further evaluation on the basis of their comparable capabilities across several key features(as highlighted in Table 4.1). These frameworks offer comparable cloud deployment capabilities, encryption support, and model management features, making them suitable candidates for the project.

- BentoML excels in its capacity for data processing, enabling the creation of customized pre- and post-processing pipelines that can be readily incorporated into a comprehensive machine learning workflow. Nevertheless, BentoML displays slightly inferior performance in comparison to other frameworks, which underscores the importance of assessing the potential implications for overall system latency.
- In comparison, TensorFlow Serving and Triton demonstrate superior performance, having been optimized for low latency inference that is a critical requirement for this project. Their high scalability and low latency represent significant advantages, particularly in the context of TensorFlow models.

4.4 Overcoming Frameworks' Limitations with FastAPI

4.4.1 TensorFlow Serving

FastAPI is chosen to overcome TensorFlow Serving's limitations and acts as an intermediary between external clients and the TensorFlow Serving, ensuring secure and efficient communication and managing authentication and request validation.

- **Separation of Concerns**: The principle of separation of concerns is a key reason for choosing FastAPI for pre- and post-processing. By isolating these tasks from the model itself, the model can remain focused solely on making predictions based on input data, without the additional burden of processing. This approach enhances the efficiency of the model, simplifies the system architecture, and improves maintainability, ensuring that each component of the system remains clearly defined and manageable[63].
- **Avoiding Custom TensorFlow Serving Handlers**: Although it is technically feasible to implement pre- and post-processing within TensorFlow Serving using custom handlers, this approach was deliberately avoided due to a number of inherent disadvantages. The creation of custom handlers would necessitate the development of C++ code and the frequent recompilation of TensorFlow Serving whenever changes are made to the processing logic[75]. Such an approach introduces significant complexity and also necessitates substantial maintenance overhead. In contrast, FastAPI provides a more flexible and readily modifiable environment, wherein alterations to the processing logic can be rapidly implemented and tested without the necessity of rebuilding TensorFlow Serving.
- **Addressing Security and Authentication**: A further notable shortcoming of TensorFlow Serving is its absence of built-in support for OAuth or alternatives to this authentication mechanism. FastAPI effectively overcomes this limitation by providing security features, including the integration of OAuth2. By managing authentication at the client level, FastAPI guarantees that only authorised requests are transmitted to TensorFlow Serving, thereby enhancing the overall security of the system. This capability is of paramount importance for the protection of sensitive data and the maintenance of secure operations in a production environment.

4.4.2 Triton Inference Server

Similarly to TensorFlow Serving, Triton Inference Server is constrained in its ability to process pre- and post-processing tasks, and lacks integrated authentication protocols. In light of these limitations, FastAPI was selected over Triton Inference Server's custom Python backend. FastAPI not only streamlines the pre- and post-processing tasks but also incorporates important security features such as OAuth2 authentication, which Triton Inference Server lacks. This guarantees that only authorised requests are routed to

Triton Inference Server, thereby offering a more secure and flexible solution without the necessity of modifying Triton Inference Server's backend for each update.

4.4.3 BentoML

Although BentoML is highly effective in providing flexible model serving and custom pre- and post-processing pipelines, it lacks the essential built-in support for authentication mechanisms, which are of paramount importance for secure operations in production environments. To address these limitations, FastAPI was selected, offering authentication via OAuth2 and guaranteeing encrypted communication using HTTPS. With HTTPS in place, all communication between the client and the server is encrypted, preventing unauthorised access to sensitive data. Additionally, OAuth2 authentication ensures that only authorised clients can access BentoML's inference services, further enhancing security.

4.5 Deployment Decisions

Once the primary frameworks for model inference had been selected, the next critical decision was to determine the most effective deployment strategy.

Both TensorFlow Serving[77] and Triton Inference Server[67] recommend the utilisation of Docker images for the deployment of models, due to the fact that Docker is capable of providing a consistent and isolated environment. Such consistency is vital for guaranteeing compatibility across diverse platforms and simplifying the deployment process. Additionally, BentoML facilitates Docker-based deployments[3], thereby establishing Docker as the optimal choice for uniformity across all frameworks. Similarly, even in the FastAPI, Docker is employed to containerize the service, thereby maintaining consistency in the deployment environment across all frameworks. The utilisation of Docker images for TensorFlow Serving, Triton Inference Server and BentoML ensures that the deployment environment remains consistent.

All major cloud providers, including Amazon Web Services (AWS), Google Cloud, and Microsoft Azure, offer comprehensive support for Docker hosting, making them suitable for the deployment of containerized applications[79]. However, Azure was selected for this project for two main reasons. First, Azure offers free credits for students, making it a more cost-effective option during the development and testing phases[1]. Second, familiarity with the Azure platform facilitated a smoother deployment process.

Additionally, Azure provides services, such as Azure Container Registry (ACR) for managing Docker images and Azure Container Instances (ACI) for efficiently running containers[10].

4.6 System Design for TensorFlow Serving with FastAPI and Triton Inference Server with FastAPI

This section describes two different designs for TensorFlow Serving and Triton Inference Server. Although these two frameworks handle model inference differently, the rest of the system architecture remains the same. The design makes use of FastAPI for data processing and security, Docker Compose for orchestration.

4.6.1 Components

FastAPI

- **Role**: FastAPI serves as the principal point of communication between external clients and the model inference server, whether TensorFlow Serving or Triton Inference Server. It is responsible for handling all incoming requests, providing security, performing the essential preprocessing, transmitting data for inference, and then postprocessing the predictions before sending them back to the client.
- **Preprocessing**: The FastAPI framework performs a preprocessing stage in which the input data is parsed, normalized and restructured in accordance with the expected format of the model in use.
- **Model Inference**:
  - **TensorFlow Serving**: In this setup, FastAPI sends the preprocessed data to TensorFlow Serving via REST API calls.
  - **Triton Inference server**: In this setup, FastAPI sends the preprocessed data to Triton Inference Server via REST API calls.

- **Postprocessing**: Once inference server has returned the predictions, FastAPI denormalizes the results, transforming them back to a meaningful scale. This post-processing step serves to ensure that the output is user-friendly and ready for interpretation.
- **Security**: FastAPI plays a pivotal role in guaranteeing the security of the system. OAuth2 authentication is employed to regulate user access, ensuring that only duly authenticated clients are permitted to interact with the API. As part of the authentication process, JWT are employed to securely transmit information between the client and the server. JWT tokens provide a means of verifying client identities, thereby enabling the server to authenticate requests without storing session information. Furthermore, FastAPI enforces HTTPS, ensuring that all communication between clients and the server is encrypted. This prevents data in transit from being intercepted or tampered with, thereby further enhancing the security of the system.

Docker Compose

- **Role**: Docker Compose is employed to orchestrate the deployment of FastAPI and whether TensorFlow Serving or Triton Inference Server as containerized services. The Docker Compose file, or docker-compose.yml, contains the configuration specifications for both frameworks, including the network settings and dependencies. Docker Compose provides a convenient means of managing the lifecycle of services, facilitating local development and production deployments. Additionally, access to TensorFlow Serving or Triton Inference Server can be limited by configuring Docker networks, ensuring that clients can only communicate with FastAPI, effectively hiding internal services like TensorFlow Serving from direct client access.

Figure 4.1 presents the system architecture designed for TensorFlow Serving/Triton Inference Server with FastAPI. The architecture demonstrates how FastAPI serves as the main interface for authentication, JWT token generation and data processing and TensorFlow Serving/Triton Inference Server performs the inference.

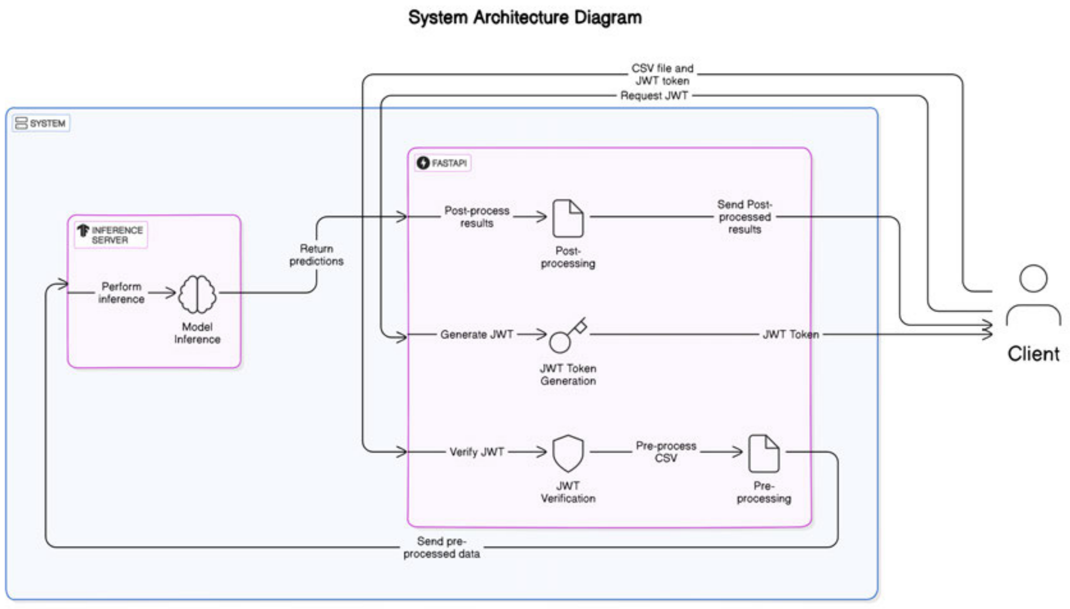


Figure 4.1: System Architecture Diagram for TensorFlow Serving/Triton Inference Server with FastAPI setup

4.6.2 Data Flow and Processing

The data flow within the system is designed in such a way as to ensure the efficient handling of client requests, from the initial ingestion of data to the subsequent inference of models and the generation of responses. Figure 4.2 illustrates the entire client-server interaction flow. The sequence of operations is as follows:

- **Token Generation**: The client begins by sending credentials to the `/token` endpoint. Following successful authentication, FastAPI generates a JWT and returns it to the client.
- **Client Request Handling**: The client includes the JWT in the Authorization header of all subsequent HTTPS requests sent to FastAPI. FastAPI verifies the token and continues processing the request.
- **Preprocessing**: The FastAPI framework reads the incoming data and applies a series of preprocessing steps with the objective of transforming the raw input into a format that is suitable for the model. As shown in Figure 4.2, the preprocessing

stage prepares the data for TensorFlow Serving/Triton Inference Server to perform inference.

- **Model Inference**: The preprocessed data is transmitted to the Inference Server via a REST API call. Subsequently, Inference server processes the data utilising the deployed machine learning model, thereby generating predictions.
- **Postprocessing and Response Generation**: FastAPI retrieves the predictions and subsequently applies postprocessing to convert the normalised predictions back to their original scale. The final results are then returned to the client as an HTTPS response.

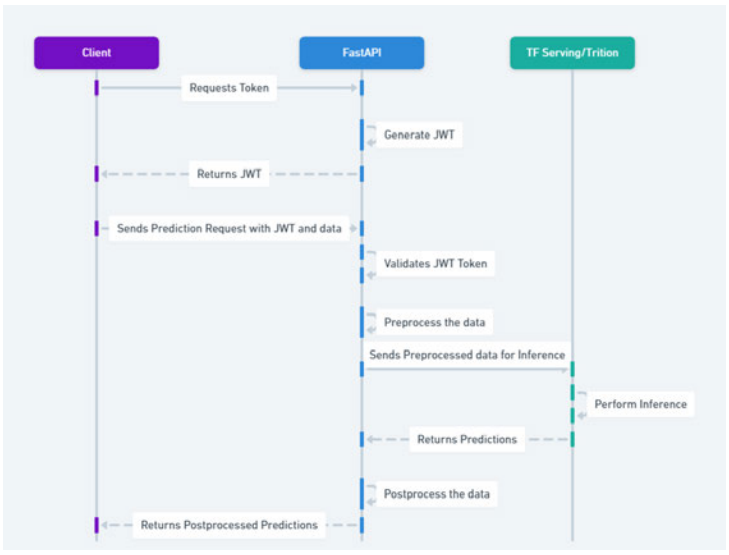


Figure 4.2: Client-Server Interaction Flow for TensorFlow Serving/Triton Inference Server Setup

4.7 System Design for Bentoml with FastAPI

The system design for BentoML follows a similar approach to the design described in Section 4.6 for TensorFlow Serving/Trition Inference Server, with FastAPI now responsible primarily for handling security tasks such as authentication and HTTPS encryption. The main difference is that in this setup, BentoML manages not only the model inference process but also the preprocessing and postprocessing stages. This allows BentoML to handle the entire data pipeline internally, while FastAPI ensures secure interaction between the client and BentoML.

4.7.1 Components

Differences in System Design Compared to TensorFlow Serving and Triton (Ch 4.6)

- **Pre- and Post-Processing**: Unlike TensorFlow Serving and Triton, which rely on external services (like FastAPI) for pre- and post-processing, BentoML incorporates these tasks internally. This leads to a more streamlined process for data handling before and after model inference within the BentoML service itself.
- **FastAPI's Role**: In contrast to Section 4.6, where FastAPI handles both API management and interaction with the model-serving framework (TensorFlow Serving or Triton), FastAPI in this design focuses exclusively on security functions such as authentication and managing HTTPS encryption. It no longer handles pre- or post-processing, leaving these operations to BentoML.
- **Deployment Setup**: Similar to the design for TensorFlow Serving and Triton, both FastAPI and BentoML are deployed as containerized services using Docker. The main difference lies in FastAPI's limited role, which is now focused solely on security while BentoML handles the full spectrum of data processing and inference.

Figure 4.3 presents the system architecture designed for BentoML with FastAPI. The architecture demonstrates how FastAPI serves as the main interface for authentication, JWT token generation and BentoML performs the inference and data processing.

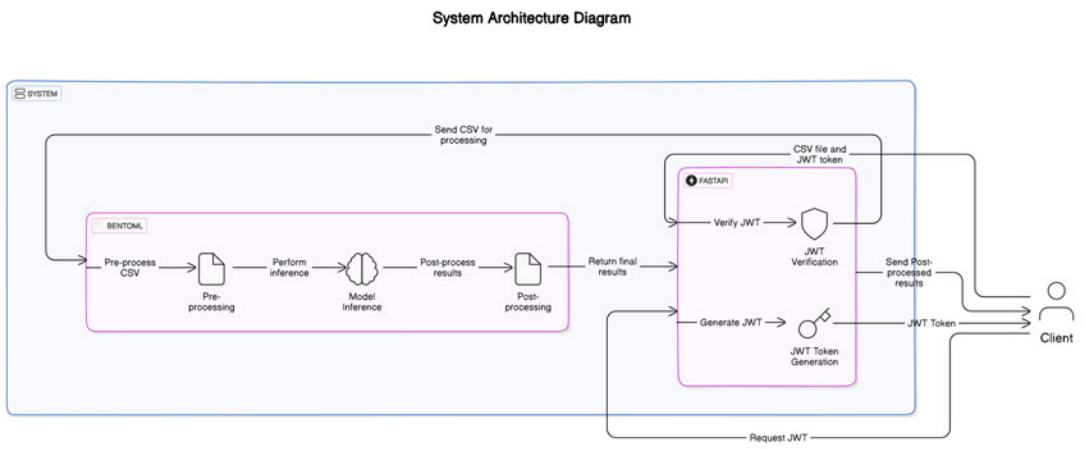


Figure 4.3: System Architecture Diagram for BentoML with FastAPI Setup

4.7.2 Data Flow and Processing

The data flow within the system is designed in such a way as to ensure the efficient handling of client requests, from the initial ingestion of data to the subsequent inference of models and the generation of responses. Figure 4.4 illustrates the entire client-server interaction flow. The sequence of operations is as follows:

- **Token Generation**: The client sends credentials to the /token endpoint. FastAPI authenticates the credentials, generates a JWT, and sends it back to the client.
- **Client Request Handling**: The client includes the JWT in the Authorization header of all subsequent HTTPS requests sent to FastAPI. FastAPI validates the token and forwards the request to BentoML.
- **Preprocessing**: Upon receiving the request, BentoML processes the input data using a series of preprocessing steps. This stage transforms the raw input into a format suitable for the machine learning model.
- **Model Inference**: After preprocessing, BentoML performs inference using the deployed model. It processes the transformed input data and generates predictions based on the model's parameters.
- **Postprocessing**: After the model inference, BentoML applies postprocessing to convert the raw predictions into a readable or interpretable format. The processed

result is then sent back to FastAPI, which generates the final response and returns it to the client as an HTTPS response.

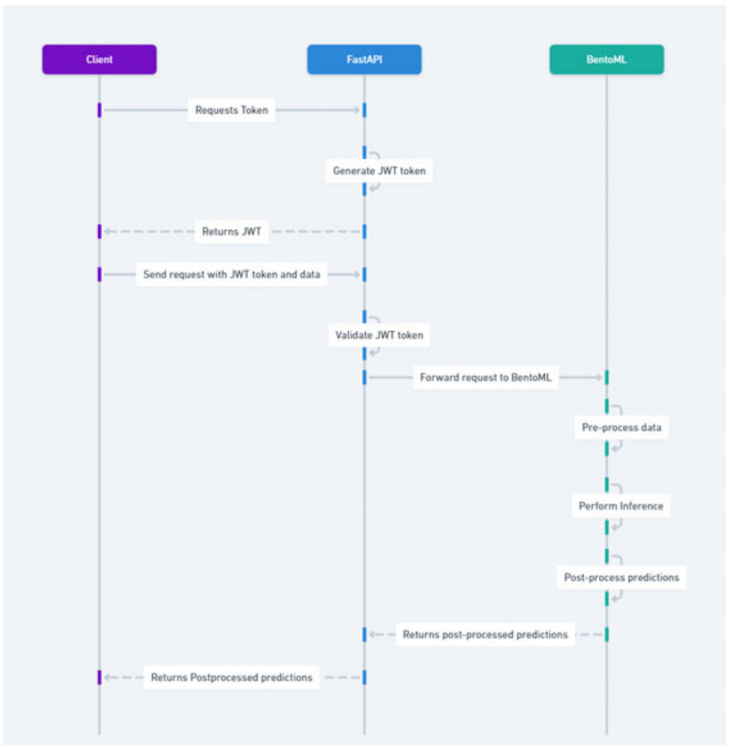


Figure 4.4: Client-Server Interaction Flow for BentoML with FastAPI setup

4.8 Cloud Deployment Setup

While the design for each framework uses different serving frameworks, the overall cloud deployment strategy remains consistent across all candidates. The system is designed to leverage cloud-based deployment using Microsoft Azure. The deployment workflow begins with storing and managing Docker images for the chosen design in ACR. Once the Docker images are uploaded, Docker Compose will orchestrate the deployment of these containerized services across the cloud environment. As depicted in Figure 4.5, the flow diagram illustrates the interaction between the developer and Azure services, detailing each step in the deployment process. The developer builds and tags a Docker image, pushes it to ACR, and then triggers the deployment using Docker Compose. ACI pull the Docker image from ACR and run the container.

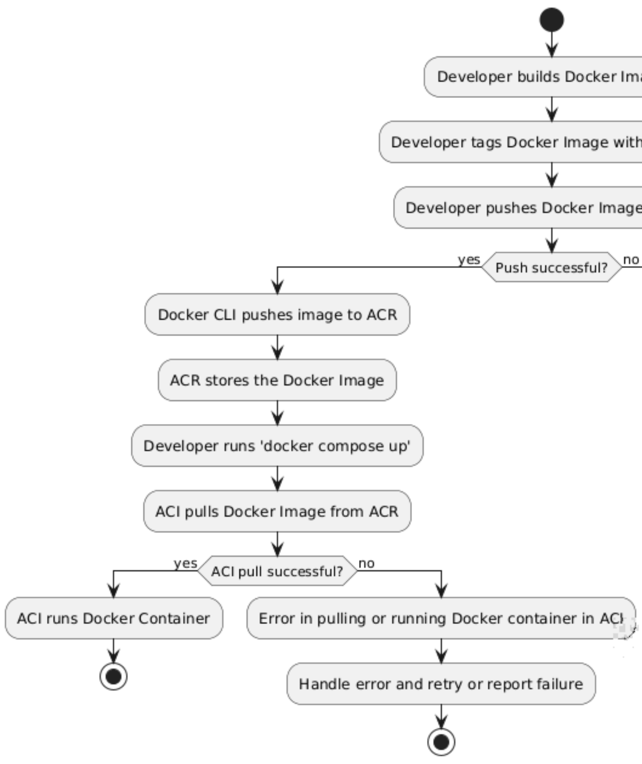

Developer builds Docker Image
Developer tags Docker Image with ACR URL
Developer pushes Docker Image to ACR
yes
Push successful?
no
Docker CLI pushes image to ACR
ACR stores the Docker Image
Developer runs 'docker compose up'
ACI pulls Docker Image from ACR
yes
ACI pull successful?
no
ACI runs Docker Container
Error in pulling or running Docker container in ACI
Handle error and retry or report failure
Error in pushing Docker image
Handle error and retry or report failure

Figure 4.5: Developer Interaction with Azure

This chapter sets forth the principal design decisions that constitute the basis of this thesis. Three candidate frameworks were selected for model inference: TensorFlow Serving, Triton, and BentoML. FastAPI plays a crucial role in these designs, serving different purposes depending on the framework. In the case of TensorFlow Serving and Triton Inference Server, FastAPI is responsible for both security and data processing, handling tasks such as authentication, preprocessing, and postprocessing of data. On the other hand, for BentoML, FastAPI is used primarily for security, managing authentication and ensuring secure access to the model inference API.

The utilisation of Docker for containerization guarantees a uniform and uncomplicated deployment process across diverse environments. Furthermore, Docker Compose is employed to orchestrate and oversee the deployment of multiple containers, ensuring seamless integration between components such as FastAPI and the inference servers. Microsoft Azure was selected as the cloud provider, with the deployment of the system being facilitated by ACI and the management of Docker images being enabled by ACR.

The final selection of the optimal serving framework will be based on performance evaluations conducted following the implementation phase. This strategic approach guarantees that the system will satisfy the project's requirements for security and performance.

5 Implementation

This chapter describes the detailed implementation of the proposed system, based on the design decisions outlined in the previous chapter. The implementation comprises three main approaches for serving machine learning models using FastAPI with TensorFlow Serving, Triton Inference Server, BentoML. Each section outlines the key steps taken in the development of the solution, including configuration, setup, and deployment. The deployment on Microsoft Azure is also discussed.

5.1 TensorFlow Serving with FastAPI

This section describes the practical steps taken to deploy TensorFlow Serving with FastAPI, using Docker for containerization and Docker Compose for orchestration.

5.1.1 Docker Configuration for TensorFlow Serving

A custom Docker image was created for TensorFlow Serving, embedding the pre-trained machine learning models directly into the image. This method eliminates the need for model mounting at runtime, simplifying the deployment process. The configuration of Docker for TensorFlow Serving was largely straightforward and did not present any significant difficulties.

This Dockerfile utilizes the official TensorFlow Serving image and copies the saved TensorFlow model into the `/models/modelname/1` directory. The `MODEL_NAME` environment variable is used to reference the model when the container is running.

5.1.2 FastAPI Client Implementation

The FastAPI application serves as the interface for users to interact with TensorFlow Serving for model inference. To ensure secure communication between the client and the FastAPI service, TLS is implemented, providing encrypted communication over HTTPS and Oauth2 for authentication. A custom Dockerfile was also created for the FastAPI, ensuring the application could be easily containerized and deployed alongside TensorFlow Serving.

Authentication Process

- **OAuth2 Password Flow**: The authentication system is based on the OAuth2 password flow. Users provide their credentials (username and password) to the `/token` endpoint to obtain a JWT (JSON Web Token). This token serves as an identification mechanism and must be included in all subsequent requests to protected API endpoints. Upon successful verification, the server responds with a JWT token, which is valid for a limited duration.
- **User Authentication**: The system validates the user's credentials by checking them against a registered user database. Passwords are stored securely using a hashing mechanism. Once authenticated, the server generates a JWT token for the user, which includes important details like the username and an expiration time.
- **Token Generation**: After successful authentication, the server creates a JWT token. The token contains a payload with the user's identification (username) and is signed with a secret key. The token is time-limited, meaning it expires after a certain period, requiring the user to log in again.
- **Token Validation**: When accessing protected resources, the user must provide the JWT token. The server validates the token by checking its signature and ensuring that it hasn't expired. If the token is valid, the user is authorized for the requested action. If the token is missing, invalid, or expired, access is denied.
- **Checking User Status**: In addition to token validation, the system checks whether the user is active. Even if a token is valid, inactive or disabled users are denied access. This provides additional control over user permissions.
- **Requesting a Token**: To start the authentication process, the user sends a request to the token endpoint, providing their login credentials. If successful, the server

generates a JWT token and returns it to the user. This token must be included in the Authorization header for all subsequent requests to the API.

The security implementation, including handling OAuth2 with JWT tokens, is based on the code samples provided in the FastAPI documentation[58].

TLS Encryption:

- FastAPI is deployed using TLS certificate, ensuring that the service runs over HTTPS on port 443.
- This ensures that any data exchanged between the client and the server, including authentication requests and prediction results, is securely transmitted and protected from unauthorized access.

Preprocessing The preprocessing stage is designed to clean, normalize, and prepare time-series data for model inference. It involves several key steps:

- **Data Loading and Cleaning**: The raw input data is loaded from a CSV file and converted into a Pandas DataFrame. During this step, columns are renamed for consistency, such as converting `Datum` to `Date` and the power consumption field to `Power Consumption [kW]`.
- **Adding Time Features**: Time-based features are added using sinusoidal and cosinusoidal transformations to capture cyclical patterns (e.g., hour, day, week, month, year). This helps improve the model's ability to detect trends over time.
- **Data Normalization**: Each column in the dataset is normalized using pre-calculated means and standard deviations stored in dictionaries. This ensures that the data matches the format used during model training, thus improving the accuracy of the inference.
- **Data Reshaping**: The processed data is reshaped into a three-dimensional format to align with the expected input structure of the model. This includes adjusting for batch size, time steps, and features.
- **Output**: The preprocessed data is prepared in a format suitable for the prediction model and is passed forward for inference.

Postprocessing The postprocessing stage takes the raw model predictions and transforms them into meaningful values that the end-user can interpret.

- **Denormalization**: The predictions are adjusted back to their original scale using the global mean (`MEAN_POST`) and standard deviation (`STD_POST`). This step is essential for converting the normalized outputs into real-world values.
- **Result Formatting**: The denormalized predictions are formatted as a list to ensure the results are easy to interpret and can be delivered to the user via the API.

The preprocessing stage is designed to clean, normalize, and prepare time-series data for model inference.

5.1.3 Deployment with Docker Compose

The system's components TensorFlow Serving and FastAPI are orchestrated using Docker Compose. The `docker-compose.yml` file connects these two services over a private network, ensuring secure and seamless communication. Both components are containerized and run in a secure environment.

- FastAPI is exposed via HTTPS on port 443 and secured using TLS certificate.
- TensorFlow Serving operates internally and is accessible only by FastAPI via the internal Docker network.

5.2 Triton Inference Server with FastAPI

In this approach, Triton Inference Server replaces TensorFlow Serving. The FastAPI is configured similarly to the TensorFlow Serving setup.

5.2.1 Docker Configuration for TensorFlow Serving

A custom Docker image for Triton Inference Server was created, embedding the TensorFlow model in the SavedModel format. Similar to TensorFlow Serving, Triton automatically loads the model during startup.

However, setting up Triton Inference Server proved to be more complex compared to TensorFlow Serving, primarily due to Triton's stringent folder structure requirements. Each model needed to be placed in a directory named after the model, with a version-numbered subfolder (e.g., 1) containing the model.savedmodel folder. This folder included the necessary model files, including saved_model.pb, variables, and assets. Additionally, configuring Triton required the creation of a config.pbtxt file to define the model's inputs and outputs. The most challenging aspect of the setup was ensuring the correct folder structure, as any misconfiguration would result in Triton failing to load the model. Moreover, the manual creation of the config.pbtxt file required detailed knowledge of the model's input and output tensors, which necessitated careful inspection of the model and increased the setup time. After several attempts both the folder layout and configuration were correctly formatted for Triton to load the model successfully.

5.2.2 FastAPI Client Implementation

The FastAPI client interacts with Triton's REST API for predictions. The request URL is updated to match Triton's inference endpoint.

Prediction Request in FastAPI:

```
url = 'http://triton:8000/v2/models/tritonsupermarket/infer'
```

Listing 5.1: Triton Inference Server URL

The rest of the functionality including preprocessing, postprocessing, and Oauth2 based authentication remains consistent with the TensorFlow Serving setup.

5.2.3 Deployment with Docker Compose

The system's components Triton Inference Server and FastAPI are orchestrated using Docker Compose. The `docker-compose.yml` file connects these two services over a private network, ensuring secure and seamless communication. Both components are containerized and run in a secure environment.

5.3 BentoML with FastAPI

This section describes the practical steps taken to deploy BentoML with FastAPI, using Docker for containerization and Docker Compose for orchestration.

5.3.1 Docker Configuration for BentoML

A custom Docker image was created for BentoML, embedding the pre-trained machine learning model directly into the image. This method eliminates the need for model mounting at runtime, simplifying the deployment process.

The Dockerfile copies the service definition (service.py) and the trained machine learning model into the container. The service was initially set up and tested locally to ensure it functioned correctly before containerization. By embedding both the service logic and the model files into the Docker image, BentoML could serve the model immediately upon startup, minimizing manual configuration.

However, several challenges were encountered during the process. While the service operated smoothly in a local environment, errors arose after containerization. BentoML handles file inputs differently in Docker, using temporary file objects that are not compatible with some standard file handling methods. This caused issues when processing files, which required adjustments to ensure compatibility across environments. Despite these complexities, the deployment was successfully completed by adapting the file handling logic for the containerized environment.

5.3.2 FastAPI Client Implementation

The FastAPI client interacts with the BentoML service for model inference. The FastAPI implementation for authentication, API routing, and TLS encryption follows the same approach as described in the TensorFlow Serving implementation in Section 5.1.2. FastAPI handles user authentication, security, and API routing, providing a consistent interface for accessing the machine learning model.

5.3.3 Preprocessing and Postprocessing

The preprocessing and postprocessing steps in BentoML follow the same structure as the TensorFlow Serving setup described in Section 5.1.2. The preprocessing function is responsible for cleaning, normalizing, and preparing the data for model inference, while the postprocessing function denormalizes and formats the predictions for user consumption.

The procedures for both preprocessing and postprocessing in BentoML are identical to those employed in the TensorFlow Serving approach. BentoML, however, handles the inference and processing directly within its service, ensuring consistent data flow and security mechanisms as previously outlined.

5.3.4 Deployment with Docker Compose

The system's components BentoML and FastAPI are orchestrated using Docker Compose. The `docker-compose.yml` file connects these two services over a private network, ensuring secure and seamless communication. Both components are containerized and run in a secure environment.

- FastAPI is exposed via HTTPS on port 443 and secured using a TLS certificate.
- BentoML operates internally and is accessible only by FastAPI via the internal Docker network.

5.4 Azure Deployment

The system is deployed on Microsoft Azure using Azure Container Instances (ACI). Azure Container Registry (ACR) is used for storing Docker images, and Docker Compose is used for orchestrating the containers.

5.4.1 Azure Resource Setup

- **Resource Group**: Created a resource group named fastapitfserving_group.
- **ACR**: Created a registry named fastapitfserving for storing Docker images.

5.4.2 Pushing Docker Images to ACR

Docker images for TensorFlow Serving and FastAPI are tagged and pushed to ACR.

```
docker tag custom_tf_serving:latest fastapitfserving.azurecr.io/
    custom_tf_serving_latest
docker tag custom_fastapi:latest fastapitfserving.azurecr.io/
    custom_fastapi_latest

az acr login --name fastapitfserving
docker push fastapitfserving.azurecr.io/custom_tf_serving_latest
docker push fastapitfserving.azurecr.io/custom_fastapi_latest
```

Listing 5.2: Tagging and Pushing Docker Images to Azure Container Registry

5.4.3 Deployment with Docker Compose

Docker Context is configured to manage the ACI directly from the Docker CLI.

```
docker context create aci fastapitfservingacicontext
docker context use fastapitfservingacicontext
docker compose up
```

Listing 5.3: Docker Commands to Create and Use Context, and Deploy with Compose

This command deploys both the FastAPI and TensorFlow Serving containers on ACI, making them accessible via HTTPS. The Docker Azure deployment process for multiple docker containers was implemented based on the tutorial provided by Patrick Koch[27].

This chapter describes the implementation of the machine learning inference system using different approaches: TensorFlow Serving, Triton Inference Server, BentoML. The system utilizes OAuth2 authentication with JWT tokens to secure API endpoints and ensure that only authorized users can interact with the system. The implementation leverages Docker for containerization and Azure for cloud deployment. The next chapter will evaluate the system's performance and security.

6 Evaluation and Testing

In this chapter, the three implemented designs TensorFlow Serving, Triton Inference Server, and BentoML with FastAPI will be tested for performance, primarily based on latency. Following the performance evaluation, a final choice of the framework will be made. The selected system will then undergo security testing to ensure robustness in real world deployment. Finally, a requirements fulfillment check will be conducted to verify that the system meets all the functional and non-functional criteria outlined earlier.

6.1 Inference Evaluation

The performance evaluation of the designs was conducted to assess how well each framework handles inference workloads when integrated with FastAPI.

6.1.1 Performance Comparison

The latency for each framework was measured using custom Python scripts. Each design was tested by sending 100 requests, and the total latency was recorded. This latency measurement includes not only the time spent on preprocessing, model inference, and postprocessing, but also the time required for authentication, as each request includes JWT token that must be validated by the system. This comprehensive evaluation provides a full picture of how these systems manage real world inference loads, accounting for the time spent on data handling and security measures. By including the time taken for token validation in each request, we obtain a more realistic evaluation of how these systems perform in secure environments, where each inference request requires authenticated access.

Metric	**TF Serving**	**Triton**	**BentoML**
Mean Latency (s)	2.271	5.327	3.456
Min Latency (s)	1.556	4.331	1.733
Max Latency (s)	3.822	7.971	6.372
Standard Deviation (s)	0.559	0.769	1.099

Table 6.1: Latency comparison across TensorFlow Serving, Triton Inference Server, and BentoML for 100 requests

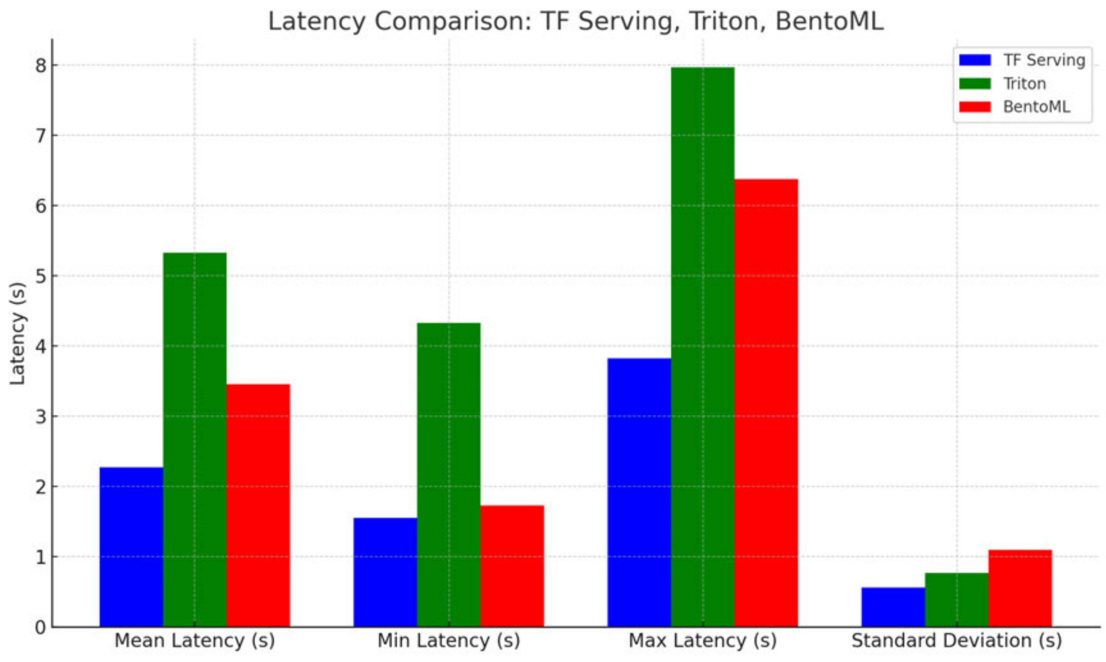


Figure 6.1: Latency Comparsion: TF Serving, Triton and BentoML

TensorFlow Serving with FastAPI exhibited the lowest latency, even when accounting for the time required for authentication and token validation. As previously discussed in the evaluation of frameworks, TensorFlow Serving is highly optimized for TensorFlow models, benefiting from deep integration within the TensorFlow ecosystem. Its architecture minimizes overhead by focusing solely on the efficient inference of TensorFlow models, without the added complexity of supporting multiple frameworks. This optimization, along with the lightweight nature of FastAPI's preprocessing and token validation, ensures consistently low latency throughout the system. As shown in Table 6.1, TensorFlow Serving consistently outperformed other frameworks, exhibiting both lower latency and lower performance variability.

On the other hand, Triton Inference Server with FastAPI demonstrated higher latency compared to TensorFlow Serving. This evaluation was conducted using a CPU-only infrastructure, whereas Triton is known to deliver significant performance improvements when deployed in GPU-accelerated environments. The lack of GPU acceleration likely contributed to the higher observed latency in this evaluation. While authentication and preprocessing overhead are present, they are secondary factors compared to the architectural complexity and the absence of GPU optimization[34].

Similarly, BentoML with FastAPI also exhibited relatively higher latency. This can largely be attributed to BentoML's design philosophy, which emphasizes extensibility and flexibility. As discussed in previous sections, BentoML allows users to define complex, custom preprocessing and postprocessing pipelines. While this flexibility is advantageous for handling diverse workflows and custom data transformations, it introduces significant overhead, contributing to the overall latency. The inclusion of token validation as part of the authentication process also adds to the latency, but the primary factor behind BentoML's longer inference times is its emphasis on user-defined processing logic.

6.1.2 Final Framework Selection

Based on the results of the latency tests and performance evaluations conducted in this chapter, TensorFlow Serving with FastAPI setuphas been selected as the most appropriate setup for this project. As shown in Table 6.1, the tests demonstrated its consistent ability to handle inference requests with the lowest latency among the frameworks tested. This decision is supported by the performance results obtained during the evaluation, which clearly indicate TensorFlow Serving's superior performance in efficiently processing and serving the deployed models.

In addition to its performance, ease of implementation played a significant role in selecting TensorFlow Serving with FastAPI, as outlined in the implementation chapter 5. TensorFlow Serving offers a straightforward setup process, with detailed documentation, pre-built Docker containers, and wide support within the TensorFlow ecosystem. This ease of deployment reduces the development time required to integrate machine learning models into production environments, making it a practical choice for teams seeking to minimize operational complexity without sacrificing performance.

Finally, TensorFlow Serving's popularity, discussed in Section 4.2.9, further supports its selection. As a component of the highly adopted TensorFlow framework, TensorFlow

Serving benefits from an extensive community, active development, and long-term support. Its widespread usage in industry ensures a wealth of resources, troubleshooting support, and best practices, which make it a reliable and future-proof option for production deployments. The framework's prominence in the machine learning ecosystem provides additional confidence in its stability and longevity.

6.1.3 Further Testing of TensorFlow Serving

Since TensorFlow Serving proved to be the fastest framework in the performance comparison, further testing was conducted to determine how much time the framework would take to perform inference alone, without the additional overhead of preprocessing, postprocessing, and authentication. To achieve this, TensorFlow Serving was tested under two different communication protocols HTTP and gRPC to compare their respective latencies for direct inference.

The goal of this test was to isolate the model inference time, providing a clearer understanding of the framework's performance when only the core task of serving predictions is evaluated. By eliminating the other components of the system, the tests focused on evaluating TensorFlow Serving's raw inference capabilities and the impact of communication protocols on latency.

Protocol	**Mean Latency (seconds)**	**Standard Deviation (seconds)**
HTTP	0.6913	0.0877
gRPC	0.6179	0.0421

Table 6.2: Comparison of HTTP and gRPC Latency in TensorFlow Serving

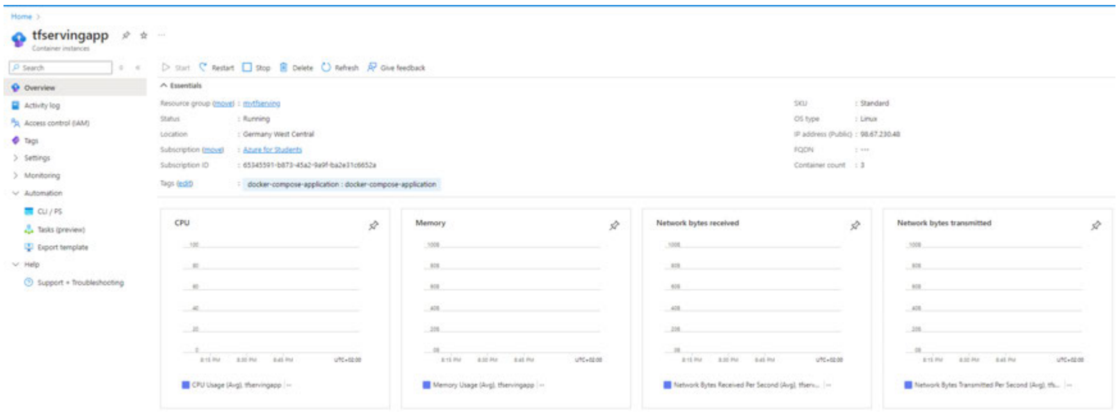
The results in Table 6.2 suggest that gRPC is superior to HTTP in terms of latency, exhibiting both lower average latency and reduced variability. Since gRPC is designed for high-performance, low-latency communication with binary serialization and persistent connections, it is typically preferred in scenarios where performance is critical. In contrast, HTTP uses text-based formats, such as JSON, which introduces additional overhead, making it less efficient than gRPC in performance-sensitive environments [70].

This also demonstrate that TensorFlow Serving exhibits extremely low inference latency, with a mean latency of 0.6179 seconds and a standard deviation of only 0.0421 seconds when using gRPC, as shown in Table 6.2. This highlights the framework's ability to deliver consistent and high performance predictions. The relatively low standard deviation indicates that TensorFlow Serving maintains stable and predictable inference performance across different requests.

However, the overall latency of the system, as presented in Table 6.1, is approximately 2.2 seconds. This higher latency is the result of additional components in the system, such as FastAPI, which handles request routing, as well as preprocessing, postprocessing, and authentication for each request. These added steps contribute to the increased latency, with the inference time itself forming only a small part of the overall latency. The overhead introduced by authentication (e.g., token validation), along with the necessary data transformations in the pre- and post-processing stages, accounts for the bulk of the total latency observed in the final system setup.

6.2 Deployed System on Microsoft Azure

The chosen system TF Serving with FastAPI was deployed on Microsoft Azure. The deployment on Azure allows for a more realistic simulation of production conditions, where cloud specific factors such as network latency, resource allocation, and geographic distribution may impact overall system performance. Figure 6.2 shows the successful deployment of the system on Azure, including resource monitoring and container instances.



Resource group (move) : mytfserving
Status : Running
Location : Germany West Central
Subscription (move) : Azure for Students
Subscription ID : 65345591-b873-45a2-9a9f-ba2e31c6652a
Tags (edit) : docker-compose-application : docker-compose-application
SKU : Standard
OS type : Linux
IP address (Public) : 98.67.230.48
FQDN : ---
Container count : 3

Figure 6.2: TensorFlow Serving with FastAPI running on Microsoft Azure, displaying resource metrics and operational status.

In order to assess the real-world performance of the chosen design, the entire system comprising FastAPI for request handling, TensorFlow Serving for model inference, and components for preprocessing, postprocessing, and authentication was deployed on Microsoft Azure. This evaluation aimed to measure the latency of the system in a cloud environment, providing a more accurate representation of its behavior under typical production conditions.

A series of 100 inference requests was sent through the system, conducted in a manner similar to the local testing. The latency for each request was recorded to assess the system's response time in a cloud setting. The results are compared with the local latency evaluation to determine the extent of any additional overhead introduced by deploying the system on a cloud platform like Azure.

The latency results are summarized in Table 6.3, which includes the mean latency, minimum and maximum latency, and the standard deviation.

Metric	**Azure Latency (seconds)**
Mean Latency	2.859
Min Latency	1.690
Max Latency	5.380
Standard Deviation	0.865

Table 6.3: Latency Results for the Deployed System on Microsoft Azure

The system deployed on Microsoft Azure experienced higher latency compared to the local environment. This increased latency can be attributed to several factors, network transmission delays, the inherent overhead of cloud infrastructure, or the geographic distance between the cloud server and the client.

6.3 Model Inference, Preprocessing and Postprocessing

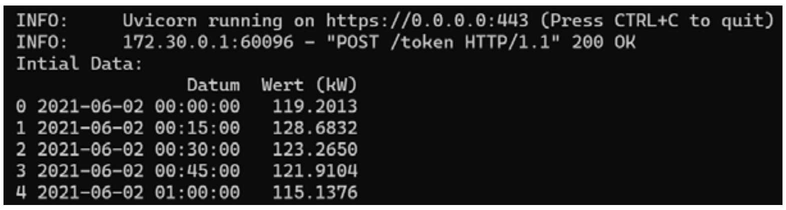
This section presents an evaluation of the system by sending a request and tracking custom outputs at each stage of the data transformation process. These outputs illustrate the various transformations as data passes through the system, including preprocessing, and postprocessing.

6.3.1 Model Inference Request and Response

A request was sent to the system. The request and response are illustrated in Figure 6.6.

6.3.2 Initial Raw Input Data

This section presents the raw data in its original form, prior to any preprocessing steps. The initial structure and content of the input data are depicted in Figure 6.3.

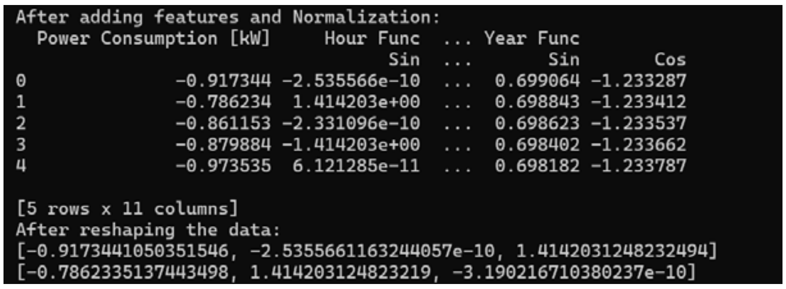


```
INFO:     Uvicorn running on https://0.0.0.0:443 (Press CTRL+C to quit)
INFO:     172.30.0.1:60096 - "POST /token HTTP/1.1" 200 OK
Intial Data:
                Datum  Wert (kW)
0 2021-06-02 00:00:00   119.2013
1 2021-06-02 00:15:00   128.6832
2 2021-06-02 00:30:00   123.2650
3 2021-06-02 00:45:00   121.9104
4 2021-06-02 01:00:00   115.1376
```

Figure 6.3: Input Data before processing.

6.3.3 Data Preprocessing

Time-based features were integrated into the data set, and normalization was applied as part of the preprocessing procedure. The resulting preprocessed data set is presented in Figure 6.4.

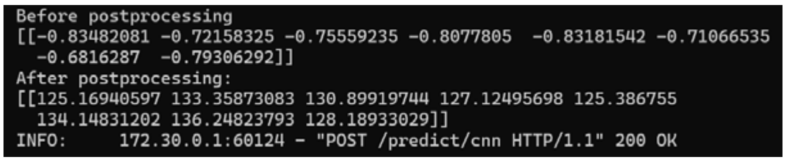


```
After adding features and Normalization:
  Power Consumption [kW]     Hour Func  ... Year Func
                                   Sin  ...       Sin       Cos
0              -0.917344 -2.535566e-10  ...  0.699064 -1.233287
1              -0.786234  1.414203e+00  ...  0.698843 -1.233412
2              -0.861153 -2.331096e-10  ...  0.698623 -1.233537
3              -0.879884 -1.414203e+00  ...  0.698402 -1.233662
4              -0.973535  6.121285e-11  ...  0.698182 -1.233787

[5 rows x 11 columns]
After reshaping the data:
[-0.9173441050351546, -2.5355661163244057e-10, 1.4142031248232494]
[-0.7862335137443498, 1.414203124823219, -3.190216710380237e-10]
```

Figure 6.4: Preprocessed Data with added time based features and normalization.

6.3.4 Data Postprocessing

The data went through postprocessing in which the results were denormalized. The final output is presented in Figure 6.5.



```
Before postprocessing
[[-0.83482081 -0.72158325 -0.75559235 -0.8077805  -0.83181542 -0.71066535
  -0.6816287  -0.79306292]]
After postprocessing:
[[125.16940597 133.35873083 130.89919744 127.12495698 125.386755
  134.14831202 136.24823793 128.18933029]]
INFO:     172.30.0.1:60124 - "POST /predict/cnn HTTP/1.1" 200 OK
```

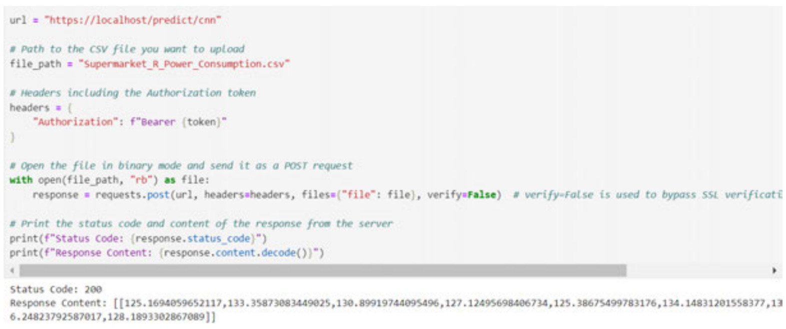
Figure 6.5: Postprocessed prediction Output.

6.4 Testing Multimodel Capability

To validate the system's capability of handling multiple models simultaneously, a manual test was conducted. This test aimed to ensure that inference requests could be routed to and processed by different models already hosted within the existing setup. Single requests were sent to their respective endpoints.

6.4.1 Request to CNN-LSTM Model

A request was manually sent to the /predict/cnn endpoint. The system successfully processed the request and returned a valid prediction from the CNN-LSTM model. The output is illustrated in the following Figure 6.6.



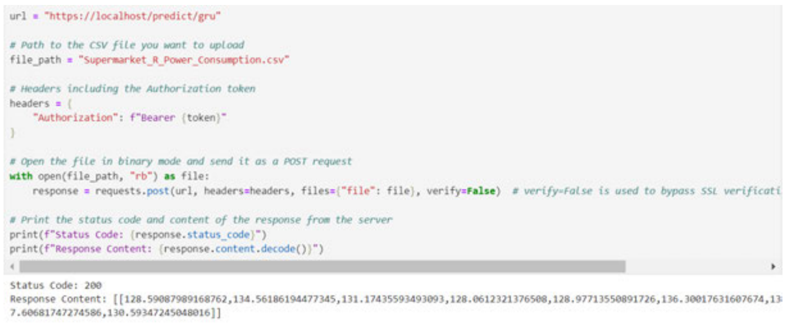
```
url = "https://localhost/predict/cnn"

# Path to the CSV file you want to upload
file_path = "Supermarket_R_Power_Consumption.csv"

# Headers including the Authorization token
headers = {
    "Authorization": f"Bearer {token}"
}

# Open the file in binary mode and send it as a POST request
with open(file_path, "rb") as file:
    response = requests.post(url, headers=headers, files={"file": file}, verify=False)  # verify=False is used to bypass SSL verificati

# Print the status code and content of the response from the server
print(f"Status Code: {response.status_code}")
print(f"Response Content: {response.content.decode()}")
```

```
Status Code: 200
Response Content: [[125.1694059652117,133.35873083449025,130.89919744095496,127.12495698406734,125.38675499783176,134.14831201558377,13
6.24823792587017,128.1893302867089]]
```

Figure 6.6: Successful POST request to the CNN-LSTM model in the TensorFlow Serving with FastAPI setup, returning a prediction response.

6.4.2 Sending Request to GRU Model

Similarly, a request was sent to the /predict/gru endpoint. The system successfully processed the request and returned a valid prediction from the GRU model. The output is illustrated in the following Figure 6.7.



```
url = "https://localhost/predict/gru"

# Path to the CSV file you want to upload
file_path = "Supermarket_R_Power_Consumption.csv"

# Headers including the Authorization token
headers = {
    "Authorization": f"Bearer {token}"
}

# Open the file in binary mode and send it as a POST request
with open(file_path, "rb") as file:
    response = requests.post(url, headers=headers, files={"file": file}, verify=False)  # verify=False is used to bypass SSL verificati

# Print the status code and content of the response from the server
print(f"Status Code: {response.status_code}")
print(f"Response Content: {response.content.decode()}")
```

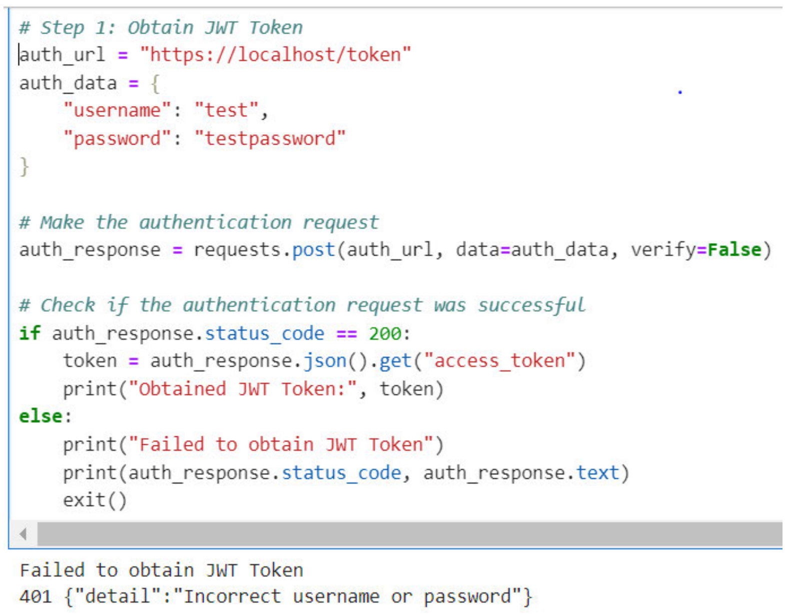
```
Status Code: 200
Response Content: [[128.59087989168762,134.56186194477345,131.17435593493093,128.0612321376508,128.97713550891726,136.30017631607674,13
7.60681747274586,130.59347245048016]]
```

Figure 6.7: Successful POST request to the GRU model in the TensorFlow Serving with FastAPI setup, returning a prediction response.

6.5 Security Testing

In order to guarantee the effectiveness and reliability of the security implementation, a series of tests were carried out.

- Providing a wrong password to the authentication system resulted in failure to generate a token, as expected. This outcome is captured in the following figure 6.8.

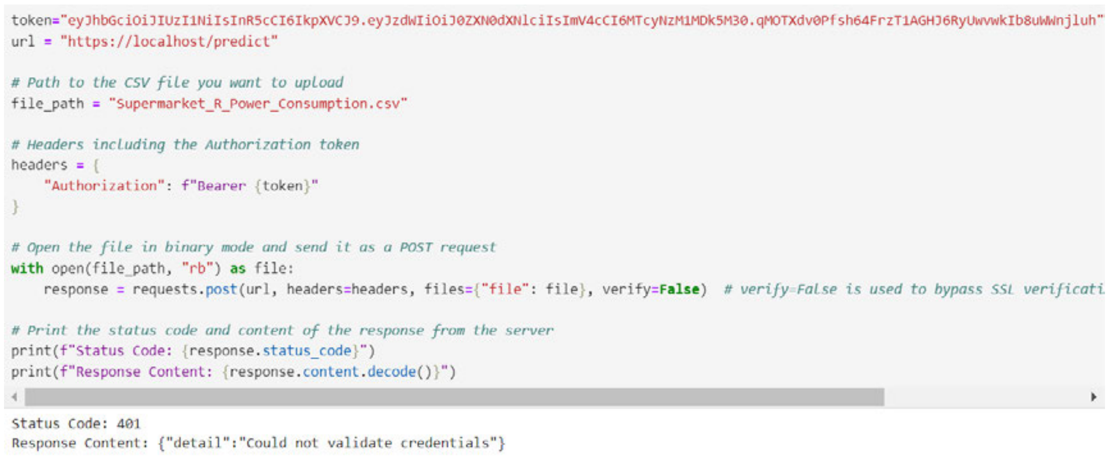


```
# Step 1: Obtain JWT Token
auth_url = "https://localhost/token"
auth_data = {
    "username": "test",
    "password": "testpassword"
}

# Make the authentication request
auth_response = requests.post(auth_url, data=auth_data, verify=False)

# Check if the authentication request was successful
if auth_response.status_code == 200:
    token = auth_response.json().get("access_token")
    print("Obtained JWT Token:", token)
else:
    print("Failed to obtain JWT Token")
    print(auth_response.status_code, auth_response.text)
    exit()
```

```
Failed to obtain JWT Token
401 {"detail":"Incorrect username or password"}
```

Figure 6.8: Request sent with incorrect credentials to the authentication endpoint, resulting in a 401 Unauthorized response and failure to obtain the JWT token.

- Requests made with an invalid JWT token were rejected with appropriate error messages. This outcome is captured in the following figure 6.9.

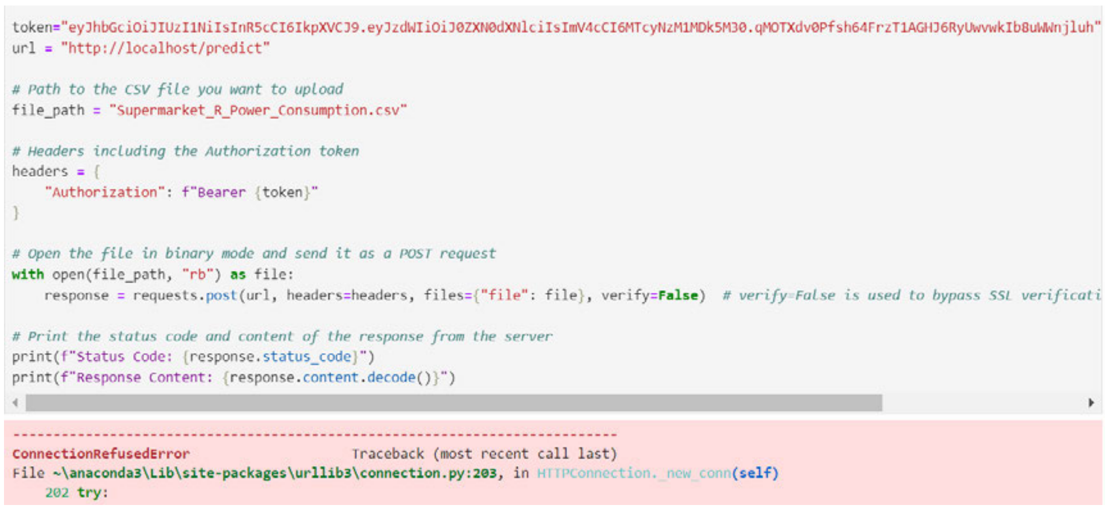


```
token="eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0ZXN0dXNlciIsImV4cCI6MTcyNzM1MDk5M30.qMOTXdv0Pfsh64FrzT1AGHJ6RyUwvwkIb8uWWnjluh"
url = "https://localhost/predict"

# Path to the CSV file you want to upload
file_path = "Supermarket_R_Power_Consumption.csv"

# Headers including the Authorization token
headers = {
    "Authorization": f"Bearer {token}"
}

# Open the file in binary mode and send it as a POST request
with open(file_path, "rb") as file:
    response = requests.post(url, headers=headers, files={"file": file}, verify=False)  # verify=False is used to bypass SSL verificati

# Print the status code and content of the response from the server
print(f"Status Code: {response.status_code}")
print(f"Response Content: {response.content.decode()}")
```

```
Status Code: 401
Response Content: {"detail":"Could not validate credentials"}
```

Figure 6.9: Request sent to the prediction endpoint with an invalid JWT token, resulting in a 401 Unauthorized response indicating the failure to validate credentials.

- Attempts to use HTTP instead of HTTPS were also blocked by the system. This outcome is captured in the following figure 6.10.



```
token="eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0ZXN0dXNlciIsImV4cCI6MTcyNzM1MDk5M30.qMOTXdv0Pfsh64FrzT1AGHJ6RyUwvwkIb8uWWnjluh"
url = "http://localhost/predict"

# Path to the CSV file you want to upload
file_path = "Supermarket_R_Power_Consumption.csv"

# Headers including the Authorization token
headers = {
    "Authorization": f"Bearer {token}"
}

# Open the file in binary mode and send it as a POST request
with open(file_path, "rb") as file:
    response = requests.post(url, headers=headers, files={"file": file}, verify=False)  # verify=False is used to bypass SSL verificati

# Print the status code and content of the response from the server
print(f"Status Code: {response.status_code}")
print(f"Response Content: {response.content.decode()}")
```

```
---------------------------------------------------------------------------
ConnectionRefusedError                    Traceback (most recent call last)
File ~\anaconda3\Lib\site-packages\urllib3\connection.py:203, in HTTPConnection._new_conn(self)
    202 try:
```

Figure 6.10: Request sent over HTTP instead of HTTPS, resulting in a ConnectionRefusedError due to the lack of secure communication protocol enforcement.

Test	Expected Outcome	Result
Invalid password	Token not generated	Passed
Invalid JWT	Access denied	Passed
HTTP request instead of HTTPS	Connection refused	Passed

Table 6.4: Security Test Results

6.6 Requirements Fulfillment Check

This section evaluates whether the system fulfills the functional and non-functional requirements outlined in Chapter 3. Each requirement is revisited, and its fulfillment is assessed based on the evaluation results presented in this chapter.

6.6.1 Functional Requirements

Table 6.5 provides an overview of the functional requirements for the system, indicating whether each requirement has been fulfilled.

FR	Description	Notes	Status
FR1	TensorFlow Model Deployment Capability	TensorFlow models were deployed	✓
FR2	Data Preprocessing	Preprocessing was implemented and tested as part of the full system evaluation.	✓
FR3	Data Postprocessing	Postprocessing was included in the full system implementation and tested.	✓
FR4	Secure Authentication	Oauth2 based authentication was implemented and tested in the security tests.	✓
FR5	API Support for Model Inference	The system supports API-based model inference	✓
FR6	Local Deployment with Containerization	Docker was used in the deployment of the system	✓
FR7	Compatibility with Cloud Providers	Microsoft Azure was used for cloud deployment.	✓
FR8	Model Management	Multiple TensorFlow Models were deployed.	✓

Table 6.5: System Requirements Fulfillment

6.6.2 Non Functional Requirements

Table 6.6 provides an overview of the non functional requirements for the system, indicating whether each requirement has been fulfilled.

NFR	**Description**	**Notes**	**Status**
NFR1	Low Latency for Request and Response	Performance tests showed low latency levels below 3 seconds.	✓
NFR2	Secure Communication	HTTPS was implemented and verified during security testing.	✓
NFR3	Framework Popularity and Support	All selected frameworks are widely used and well-supported.	✓

Table 6.6: Non Functional Requirements Fulfillment

In conclusion, TF Serving with FastAPI setup provides the best overall performance for system configurations, as well as efficient security handling. The system is now ready for deployment, with all key requirements for performance, security, and reliability being met.

7 Conclusion

In this thesis, an extensive evaluation of various model serving frameworks, such as TensorFlow Serving, Triton Inference Server, BentoML, TorchServe, and FastAPI was conducted. The evaluation was based on key criteria such as security, latency performance, ease of deployment, and multi-model support, which are critical for machine learning model deployment in real-world production environments. Initially, TorchServe and FastAPI were considered as part of the theoretical evaluation. However, after assessing their capabilities, TensorFlow Serving, Triton, and BentoML were identified as the most promising candidates for further practical evaluation.

Of the three, TensorFlow Serving demonstrated the lowest latency during both setup and testing, thereby establishing itself as the most effective performer. Furthermore, its pervasive presence within the machine learning ecosystem guarantees extensive community support and uninterrupted updates, thus reinforcing its future-proof quality. Although Triton and BentoML afforded greater flexibility, particularly in serving different types of models, they introduced additional overhead in multi-model scenarios, resulting in higher latency compared to the TensorFlow Serving and FastAPI combination. The final deployment setup was determined to be TensorFlow Serving with FastAPI, which was selected based on its superior latency performance, ease of deployment, and extensive support from the community.

The chosen system design, TensorFlow Serving is responsible solely for inference, while FastAPI manages the critical tasks of preprocessing, postprocessing, and authentication. This division of responsibilities allowed for a flexible and secure system where FastAPI efficiently handles client requests, ensuring that data is properly prepared before being passed to TensorFlow Serving and securely processed after inference. Additionally, FastAPI implements OAuth2-based authentication to safeguard model access, ensuring that only authorized users can make inference requests.

The selected system was carefully evaluated to ensure it functioned properly. Initially, the unprocessed data was fed into the system to ensure that it was able to successfully process the data and generate predictions. Secondly, the system's capacity to manage multiple models was validated through the successful processing of requests to both the CNN-LSTM and GRU models. This outcome serves to demonstrate the system's capability to handle diverse models in a simultaneously. Thirdly, the security measures in place, including OAuth2-based authentication and encryption via TLS, were evaluated to ensure the secure protection of data and model access. Finally, the system was deployed on Microsoft Azure, where it maintained low-latency inference, despite minor increases in latency due to cloud infrastructure, proving its effectiveness for cloud-based production environments.

It is important to note that the evaluation was conducted using a personal computer with only a Central Processing Unit (CPU) available. Latency may be reduced across all configurations if tested on a Graphics Processing Unit (GPU) enabled system or a more powerful CPU, which could result in faster inference times. Triton Inference Server has been specifically optimized for NVIDIA GPUs, which may result in a notable enhancement in performance when utilized in a GPU environment. While TensorFlow Serving exhibited the most favorable performance in this CPU-bound setup, the outcomes may diverge in a GPU enabled infrastructure.

In conclusion, the combination of TensorFlow Serving and FastAPI has emerged as the optimal solution for machine learning model deployment in this project. For future work, several key areas can be explored to enhance the evaluation and deployment process. The use of more powerful hardware, including GPU-enabled systems, would facilitate a better assessment of the performance potential of each framework. In addition, performance can be evaluated using a broader set of metrics than just latency, and processes such as model training and deployment of the latest models can be automated.

Bibliography

[1] Azure, Microsoft: *Azure for Students*. https://azure.microsoft.com/en-us/free/students/. 2024. – Accessed: September 4, 2024

[2] Baeldung: *OkHttp Timeouts*. 2023. – URL https://www.baeldung.com/okhttp-timeouts. – Accessed: October 1, 2024

[3] BentoML: *BentoML Containerization Guide*. https://docs.bentoml.org/en/latest/guides/containerization.html. 2024. – Accessed: September 5, 2024

[4] BentoML: *BentoML GitHub Repository*. https://github.com/bentoml/BentoML. 2024. – Accessed: September 4, 2024

[5] BentoML: *BentoML Model Composition Guide*. https://docs.bentoml.com/en/latest/guides/model-composition.html. 2024. – Accessed: September 4, 2024

[6] BentoML: *BentoML Quickstart Guide*. https://docs.bentoml.com/en/latest/get-started/quickstart.html. 2024. – Accessed: September 4, 2024

[7] BentoML: *BentoML Security Guide*. https://docs.bentoml.com/en/1.1/guides/security.html. 2024. – Accessed: September 4, 2024

[8] BentoML: *Breaking Up with Flask & FastAPI: Why ML Model Serving Requires a Specialized Framework*. https://bentoml.com/blog/breaking-up-with-flask-amp-fastapi-why-ml-model-serving-requires-a-specialized-framework. 2024. – Accessed: September 4, 2024

[9] BentoML: *Introduction to BentoML*. https://docs.bentoml.com/en/latest/get-started/introduction.html. 2024. – Accessed: September 4, 2024

[10] BlueXP: *Azure Container Instances vs. AKS: How to Choose.* https://bluexp.netapp.com/blog/azure-cvo-blg-azure-container-instances-vs-aks-how-to-choose#:~:text=Azure%20Container%20Instances%20(ACI)%20offers,AWS%20service%2C%20Amazon%20Fargate. 2024. – Accessed: September 4, 2024

[11] Community, Atlassian: *Understanding the MoSCoW Prioritization: How to Implement It into Your Workflows.* 2023. – URL https://community.atlassian.com/t5/App-Central-articles/Understanding-the-MoSCoW-prioritization-How-to-implement-it-into/ba-p/2463999. – Accessed: September 30, 2024

[12] Dagster: *Data Reshaping.* 2023. – URL https://dagster.io/glossary/data-reshaping. – Accessed: September 23, 2024

[13] Developers, NumPy: *What is NumPy?* 2023. – URL https://numpy.org/doc/stable/user/whatisnumpy.html. – Accessed: September 23, 2024

[14] Docker, Inc.: *Docker Compose.* 2023. – URL https://docs.docker.com/compose/. – Accessed: September 23, 2024

[15] Docker, Inc.: *Docker Compose: Features and Uses.* 2023. – URL https://docs.docker.com/compose/intro/features-uses/. – Accessed: September 23, 2024

[16] Docker, Inc.: *Docker Overview.* 2023. – URL https://docs.docker.com/get-started/docker-overview/. – Accessed: September 23, 2024

[17] Docker, Inc.: *What is a Container?* 2023. – URL https://www.docker.com/resources/what-container/. – Accessed: September 23, 2024

[18] FastAPI: *FastAPI Cloud Deployment Guide.* https://fastapi.tiangolo.com/deployment/cloud/. 2024. – Accessed: September 4, 2024

[19] FastAPI: *FastAPI First Steps Tutorial.* https://fastapi.tiangolo.com/tutorial/first-steps/. 2024. – Accessed: September 4, 2024

[20] FastAPI: *FastAPI GitHub Repository.* https://github.com/fastapi/fastapi. 2024. – Accessed: September 4, 2024

[21] FastAPI: *FastAPI Security Tutorial.* https://fastapi.tiangolo.com/tutorial/security/. 2024. – Accessed: September 4, 2024

[22] FRAUNHOFER-GESELLSCHAFT: *Fraunhofer Publication: Document content*. 2023. – URL https://publica-rest.fraunhofer.de/server/api/core/bitstreams/1b99fd91-d422-4ad2-871d-8e73c904be9b/content. – Accessed: September 4, 2024

[23] HARDT, Dick: *The OAuth 2.0 Authorization Framework*. RFC 6749. Oktober 2012. – URL https://www.rfc-editor.org/info/rfc6749

[24] HOPSWORKS: *What is Model Serving?* 2024. – URL https://www.hopsworks.ai/dictionary/model-serving. – Accessed: September 18, 2024

[25] HTTPX PROJECT: *Timeouts in HTTPX*. 2023. – URL https://www.python-httpx.org/advanced/timeouts/. – Accessed: October 1, 2024

[26] IGUAZIO: *Introduction to TensorFlow Serving*. https://www.iguazio.com/blog/introduction-to-tf-serving/. 2024. – Accessed: September 4, 2024

[27] KOCH, Patrick: *Docker Azure Deployment Tutorial*. https://www.patrickkoch.dev/posts/post_20/. 2022. – Accessed: September 23, 2024

[28] LINUXPIP: *Timeout in Python Requests - Everything You Need to Know*. 2023. – URL https://linuxpip.org/requests-timeout-in-python/. – Accessed: October 1, 2024

[29] LOADFORGE: *Securing Your FastAPI Web Service: Best Practices and Techniques*. https://loadforge.com/guides/securing-your-fastapi-web-service-best-practices-and-techniques. 2024. – Accessed: September 4, 2024

[30] MICROSOFT: *Introduction to Azure Container Registry*. 2023. – URL https://learn.microsoft.com/en-us/azure/container-registry/container-registry-intro. – Accessed: September 23, 2024

[31] MICROSOFT: *Overview of Azure Container Instances*. 2023. – URL https://learn.microsoft.com/en-us/azure/container-instances/container-instances-overview. – Accessed: September 23, 2024

[32] MICROSOFT: *Tutorial: Prepare Azure Container Registry for Azure Container Instances*. 2023. – URL https://learn.microsoft.com/en-us/azure/container-instances/container-instances-tutorial-prepare-acr. – Accessed: September 23, 2024

[33] NEPTUNE.AI: *How to Serve Machine Learning Models with TensorFlow Serving and Docker.* https://neptune.ai/blog/how-to-serve-machine-learning-models-with-tensorflow-serving-and-docker. 2024. – Accessed: September 4, 2024

[34] NEPTUNE.AI: *ML Model Serving: Best Tools.* https://neptune.ai/blog/ml-model-serving-best-tools. 2024. – Accessed: September 4, 2024

[35] NOVOGRODER, Idan: Data Preprocessing in Machine Learning: Steps & Best Practices. In: *LakeFS Blog* (2024). – URL https://lakefs.io/blog/data-preprocessing-in-machine-learning/. – Accessed: September 23, 2024

[36] NVIDIA: *Triton Inference Server Documentation.* https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html. 2023. – Accessed: September 23, 2024

[37] NVIDIA: *Accelerating Inference with Triton Inference Server and DALI.* https://developer.nvidia.com/blog/accelerating-inference-with-triton-inference-server-and-dali/. 2024. – Accessed: September 4, 2024

[38] NVIDIA: *Inference Protocols Customization Guide.* https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/customization_guide/inference_protocols.html. 2024. – Accessed: September 4, 2024

[39] NVIDIA: *Triton Inference Server Model Management Guide.* https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/model_management.html. 2024. – Accessed: September 4, 2024

[40] NVIDIA: *Triton Inference Server Quickstart Guide.* https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/getting_started/quickstart.html. 2024. – Accessed: September 4, 2024

[41] OAUTH: *OAuth 2.0 Authorization Framework.* 2023. – URL https://oauth.net/2/. – Accessed: September 23, 2024

[42] OAUTH: *OAuth 2.0 Bearer Tokens.* 2023. – URL https://oauth.net/2/bearer-tokens/. – Accessed: September 23, 20243

[43] OAuth: *OAuth 2.0 JWT Authorization.* 2023. – URL https://oauth.net/2/jwt/. – Accessed: September 23, 2024

[44] Overflow, Stack: *What is the Version of SSL/TLS in TensorFlow Serving?* https://stackoverflow.com/questions/51087490/what-is-the-version-of-ssl-tls-in-tensorflow-serving. 2024. – Accessed: September 4, 2024

[45] PyPI: *BentoML PyPI Stats.* https://pypistats.org/packages/bentoml. 2024. – Accessed: September 4, 2024

[46] PyPI: *FastAPI PyPI Stats.* https://pypistats.org/packages/fastapi. 2024. – Accessed: September 4, 2024

[47] PyPI: *TensorFlow Serving API PyPI Stats.* https://pypistats.org/packages/tensorflow-serving-api. 2024. – Accessed: September 4, 2024

[48] PyPI: *TorchServe PyPI Stats.* https://pypistats.org/packages/torchserve. 2024. – Accessed: September 4, 2024

[49] PyPI: *TritonClient PyPI Stats.* https://pypistats.org/packages/tritonclient. 2024. – Accessed: September 4, 2024

[50] Python Software Foundation: *Advanced Usage — Requests 2.32.3 documentation.* 2023. – URL https://docs.python-requests.org/en/latest/user/advanced/. – Accessed: October 1, 2024

[51] PyTorch: *PyTorch Serve Configuration.* https://pytorch.org/serve/configuration.html. 2024. – Accessed: September 4, 2024

[52] PyTorch: *PyTorch Serve Examples README.* https://github.com/pytorch/serve/blob/master/examples/README.md. 2024. – Accessed: September 4, 2024

[53] PyTorch: *PyTorch Serve GitHub Repository.* https://github.com/pytorch/serve. 2024. – Accessed: September 4, 2024

[54] PyTorch: *PyTorch Serve Management API Documentation.* https://github.com/pytorch/serve/blob/master/docs/management_api.md. 2024. – Accessed: September 4, 2024

[55] PyTorch: *PyTorch Serve Server Documentation.* https://pytorch.org/serve/server.html. 2024. – Accessed: September 4, 2024

[56] PyTorch: *TorchServe*. https://pytorch.org/serve/. 2024. – Accessed: September 5, 2024

[57] Qwak: *Top ML Serving Tools*. https://www.qwak.com/post/top-ml-serving-tools. 2024. – Accessed: September 4, 2024

[58] Ramírez, Sebastián: *FastAPI - OAuth2 with Password (and hashing), Bearer with JWT tokens*. https://fastapi.tiangolo.com/tutorial/security/oauth2-jwt/. – Accessed: September 23, 2024

[59] Ramírez, Sebastián: *FastAPI*. https://fastapi.tiangolo.com/. 2018. – Accessed: September 23, 2024

[60] Research, Google: *Running Your Models in Production with TensorFlow Serving*. 2023. – URL https://research.google/blog/running-your-models-in-production-with-tensorflow-serving/. – Accessed: September 28, 2024

[61] Run:AI: *Triton Inference Server Guide*. https://www.run.ai/guides/machine-learning-engineering/triton-inference-server#features. 2024. – Accessed: September 4, 2024

[62] Scaler: *TensorFlow FastAPI Integration Guide*. https://www.scaler.com/topics/tensorflow/tensorflow-fastapi/. 2024. – Accessed: September 4, 2024

[63] Sculley, D. ; Holt, Gary ; Golovin, Daniel ; Davydov, Eugene ; Phillips, Todd ; Ebner, Dietmar ; Chaudhary, Vinay ; Young, Michael ; Crespo, Jean-François ; Dennison, Dan: Hidden Technical Debt in Machine Learning Systems. In: Cortes, C. (Hrsg.) ; Lawrence, N. (Hrsg.) ; Lee, D. (Hrsg.) ; Sugiyama, M. (Hrsg.) ; Garnett, R. (Hrsg.): *Advances in Neural Information Processing Systems* Bd. 28, Curran Associates, Inc., 2015, S. 2503–2511. – URL https://proceedings.neurips.cc/paper_files/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf. – Accessed: September 4, 2024

[64] Sefidian, Amir: *Handling Cyclical Features Such as Hours in a Day for Machine Learning Pipelines with Python Example*. 2021. – URL https://www.sefidian.com/2021/03/26/handling-cyclical-

features-such-as-hours-in-a-day-for-machine-learning-pipelines-with-python-example/. – Accessed: September 23, 2024

[65] Server, Triton I.: *Triton Inference Server GitHub Repository*. https://github.com/triton-inference-server/server. 2024. – Accessed: September 4, 2024

[66] Server, Triton I.: *Triton Inference Server Model Deployment Conceptual Guide*. https://github.com/triton-inference-server/tutorials/blob/main/Conceptual_Guide/Part_1-model_deployment/README.md. 2024. – Accessed: September 4, 2024

[67] Server, Triton I.: *Triton Inference Server Quickstart Guide*. https://github.com/triton-inference-server/server/blob/main/docs/getting_started/quickstart.md. 2024. – Accessed: September 5, 2024

[68] Services, Amazon W.: *The Difference Between SSL and TLS*. 2023. – URL https://aws.amazon.com/compare/the-difference-between-ssl-and-tls/. – Accessed: September 23, 20243

[69] Services, Amazon W.: *What Is Containerization?* 2023. – URL https://aws.amazon.com/what-is/containerization/. – Accessed: September 23, 2024

[70] Services, Amazon W.: *The Difference Between gRPC and REST*. 2024. – URL https://aws.amazon.com/compare/the-difference-between-grpc-and-rest/. – Accessed: September 30, 2024

[71] Shokrzad, Reza: *FastAPI: The Modern Toolkit for Machine Learning Deployment*. https://medium.com/@reza.shokrzad/fastapi-the-modern-toolkit-for-machine-learning-deployment-af31d72b6589. 2024. – Accessed: September 4, 2024

[72] Storage, Pure: *What is Data Preprocessing?* 2023. – URL https://www.purestorage.com/knowledge/what-is-data-preprocessing.html. – Accessed: September 23, 2024

[73] Supertype.ai: *Serving PyTorch with TorchServe*. https://supertype.ai/notes/serving-pytorch-w-torchserve/. 2024. – Accessed: September 4, 2024

[74] Team, Pandas D.: *Pandas: Python Data Analysis Library*. 2023. – URL https://pypi.org/project/pandas/. – Accessed: September 23, 2024

[75] TensorFlow: *Issue #663: TensorFlow Serving SSL/TLS Mutual Authentication*. https://github.com/tensorflow/serving/issues/663. 2024. – Accessed: September 4, 2024

[76] TensorFlow: *TensorFlow Serving*. https://www.tensorflow.org/tfx/guide/serving. 2024. – Accessed: August 29, 2024

[77] TensorFlow: *TensorFlow Serving GitHub Repository*. https://github.com/tensorflow/serving. 2024. – Accessed: September 4, 2024

[78] TensorFlow: *TensorFlow Serving on Kubernetes Guide*. https://github.com/tensorflow/serving/blob/master/tensorflow_serving/g3doc/serving_kubernetes.md. 2024. – Accessed: September 4, 2024

[79] Vaughan-Nichols, Steven: A Decade of Docker. In: *Open Source Watch* (2023). – URL https://opensourcewatch.beehiiv.com/p/decade-docker. – Accessed: September 4, 2024

A Appendix

The appendix to the thesis is on CD and can be obtained from the first examiner.

Declaration

I declare that this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used.

		[redacted]
City	Date	Signature