IBM-UCL AI Islands

SYSTEM DESIGN AND INTEGRATION OF OFFLINE TEXT-GENERATION MODELS

23/09/2024

JJPL5

COMP0073: MSc Computer Science Project

Department of Computer Science
University College London

Abstract

The widespread adoption of AI technologies is often hindered by high infrastructure costs, technical complexities, and data privacy concerns. These challenges prevent smaller businesses and educational institutions from fully leveraging AI solutions, particularly for tasks such as Generative AI for text generation and chatbot functionalities. To address these barriers, the AI Islands system was proposed as a cost-effective, offline solution that enables users to manage and use AI models without needing extensive technical expertise or cloud services, all within an intuitive and accessible interface.

Devised as a four-member group project by IBM and UCL under the IXN banner, the primary team goal was to create a comprehensive index of offline AI models, supporting natural language, visual, and audio processing, while also integrating online functionality through IBM Watson services. By incorporating this index, the aim was to provide a flexible platform that empowers users to easily navigate and manage a range of AI models. Within this framework, the personal goal focused on researching and integrating Generative AI models into the offline index, enabling users to create chatbots with capabilities similar to IBM Watson's offerings. This report highlights how these efforts align with the overarching team goals, resulting in a scalable and adaptable solution that meets both individual and collective project aims.

The personal contributions centred on the design and implementation of the backend system, supporting a comprehensive collection of offline, open-source Generative AI text-generation models. Extensive research into AI technologies informed the integration of these models, allowing users to search, download, and manage them entirely offline. Key features such as the playground for model experimentation via model chaining allow users to customise workflows, enhancing flexibility in AI-powered operations. The backend also supports seamless integration with external applications through a robust API framework, ensuring system versatility and scalability.

The success of this solution has been demonstrated through extensive testing and positive stakeholder feedback. The project's impact is significant, with industry stakeholders at IBM and those in the educational sector recognising its potential to advance AI adoption by improving access to offline model sources. A video demonstration, focusing on text-generation models, is available at the following link: `https://youtu.be/30NjYHuSXXs`

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

Artificial intelligence (AI) has made rapid advancements in recent years, with breakthroughs in machine learning, Generative AI, data availability, and computational power enabling sophisticated models for tasks such as natural language processing, image recognition, and text-generation. Large-scale models like OpenAI's GPT and Google's BERT have expanded AI's capabilities, but they also demand vast computational resources, raising concerns about energy consumption, infrastructure efficiency, and scalability [1]. These challenges, alongside concerns about data privacy, technical complexity of integration, and high costs, have created significant barriers for smaller businesses, educational institutions, and individuals seeking to adopt AI technologies.

The AI Islands 4-member team project, developed as part of the UCL IXN programme in collaboration with IBM, aimed to address some of these challenges. The project brief, provided by Stakeholder Professor Dean Mohamedally and Industry Stakeholder Professor John Macnamara, called for a cost-effective, offline AI solution that would reduce reliance on cloud infrastructure and provide an accessible platform for managing AI models, reducing the barrier to AI access for less technically experienced users. The team's goal was to create a comprehensive index of offline AI models, supporting a range of tasks, and enabling users to run and use these models across different applications, all within an intuitive user-friendly interface. The first team member focused on natural language and audio processing models, while the second team member was responsible for integrating computer vision-related models. The third member handled the integration of IBM Watson services to provide optional online functionality, complementing the offline solution.

As the fourth member of the team, the personal project goal involved the research, design, and integration of suitable Generative AI models for text generation into the offline index. This enables users to create chatbots with functionalities similar to those offered by IBM Watson, aligning with the broader team objective of developing a scalable and adaptable system that supports a wide range of AI tasks. In addition to these contributions, serving as the project team leader added further responsibilities, including guiding the overall development process, liaising with stakeholders, and playing a key role in shaping the system's architecture and implementation. These leadership duties ensured that the system's architecture remained cohesive and scalable, effectively supporting the team's goal of building a comprehensive maintainable platform for managing AI models.

The main focus of this report is on the design and development of the backend system, with specific focus on text-generation tasks. The user interface (UI), in contrast, is discussed in the context of contributions to its requirement gathering and design, with further technical details provided when exploring its integration with the broader system.

1.2 Project Goals and Personal Aims

1.2.1 Project Goals

The team goals of the AI Islands project build on the solutions outlined in the motivation, aiming to develop a desktop AI tool that addresses key barriers, making AI accessible to a wider range of users, particularly non-experts, small businesses, and educational institutions. As team leader, the personal goals of this project are closely aligned with the broader team objectives. However, specific aspects, highlighted below, reflect desired personal outcomes that will be achieved through the successful fulfilment of the individual brief:

Lowering the Technical Barrier for AI Adoption The AI Islands project aims to create a desktop application that reduces the need for technical expertise. By offering a searchable index of offline AI models, particularly featuring Generative AI text-generation models, the tool will allow non-technical users to easily find and deploy these models without requiring deep knowledge of machine learning frameworks. The intuitive interface is designed to make sophisticated Generative AI models accessible to a wide range of users.

Improving the Affordability of AI Solutions To address the costs associated with cloud-based AI services and high-end hardware, particularly relevant for large generative AI models, the AI Islands project will provide a cost-effective alternative. The application is designed to run text-generation AI models on limited hardware, lowering infrastructure costs and making AI adoption more feasible for small businesses, educational institutions, and individual users.

Offering Privacy-Conscious AI Tools The project places a strong emphasis on offline functionality, enabling users to run AI models locally and retain full control over their data. This is particularly significant for generative AI models used in data querying, where data is often processed online. By maintaining data locally, the project addresses privacy concerns, especially for industries like healthcare and finance, where data protection is critical.

Supporting Scalability and Integration Designed to cater to a diverse user base, the application will offer API access for seamless integration with external systems. This scalability allows users, ranging from students to small businesses, to expand their AI usage and adapt the tool to various needs, all without incurring high operational costs. In particular, this will enable users to leverage the capabilities of offline text-generation models in their own applications, reducing their reliance on services like OpenAI.

Customisation and Flexibility The AI Islands project will empower users to chain smaller models together to create complex AI solutions, offering an alternative to large multimodal models. Users will also have the flexibility to configure and fine-tune models, tailoring them to specific requirements. The ability to save custom chain configurations for repeated use or integration into workflows further enhances the tool's versatility, enabling users to explore AI solutions without needing advanced technical skills.

1.2.2 Personal Aims

This project provides a valuable opportunity for both technical and leadership growth, with a focus on advancing skills in Generative AI, API development, software engineering, and Agile methodologies. It also allows for hands-on experience in managing teams and interacting with stakeholders, contributing to overall professional development.

Expand Knowledge of AI and Machine Learning A key aim is to deepen personal understanding of the AI landscape, particularly in transformer-based models that power Generative AI. By working on model curation and exploring how transformers function in natural language tasks, this

goal will enhance technical knowledge and practical experience in real-world AI applications.

Enhance Software Engineering Skills This project will strengthen software engineering abilities by developing an object-oriented backend in Python. The aim is to refine skills in writing modular, scalable code that adheres to best practices, building a solid foundation for creating maintainable and adaptable software solutions.

Develop REST API Expertise A personal objective is to gain hands-on experience in designing and integrating RESTful APIs, with a focus on MLOps patterns. This will build expertise in developing scalable and secure APIs that enable AI models to interact with external systems, a critical skill for deploying AI in enterprise environments.

Improve Understanding of Agile Frameworks and GitHub Kanban By applying Agile methodologies and using GitHub Kanban for project management, this aim is to improve personal efficiency in Agile workflows, including task prioritisation, sprint planning, and collaborative work within a team setting.

Build Leadership and stakeholder Management Skills Leading a team and liaising with stakeholders will offer direct experience in leadership and project management. The goal is to strengthen communication and organisational skills, ensuring alignment with stakeholder expectations while effectively managing the team to deliver project outcomes.

1.3 Project Management

The COMP0073: MSc Computer Science Project presented unique challenges in the development of the AI Islands system. Each team member was required to maintain a degree of separation in their feature development to meet individual project briefs, while still collaborating to build the application as a cohesive whole. This made the implementation of a strict Agile framework, such as Scrum, impractical, as it could have imposed too many restrictions on individual progress. Conversely, using Kanban alone would not have provided the daily standups needed for effective team coordination.

To address these challenges, a hybrid approach was adopted. The team used GitHub's Kanban system to track feature progress, while daily standups ensured coordination and allowed for regular feedback. This combination allowed team members to maintain the necessary flexibility in their individual work while keeping the project aligned through continuous communication.

Weekly meetings with the project supervisor, Dr. Yun Fu, provided highly valuable insight and support throughout development. Additionally, bi-weekly reviews with the project stakeholders offered important feedback, reinforcing the focus on customer collaboration that is central to Agile methodologies.

1.4 Report Structure

This section outlines the overall report structure, briefly summarising the main themes discussed in each chapter.

2. **Requirements**

   This chapter outlines the project's requirements, beginning with the individual brief and detailing how it evolved during the stakeholder discussions. It defines target personas, relevant use cases, and uses a MoSCoW analysis to prioritise the key features and functionalities necessary for the project.

3. **Research**

The Research chapter reviews existing technologies and tools, comparing them against the project's needs. It examines similar applications, evaluates model selection and sourcing, and details the technology stack for both the backend and frontend of the application.

4. **HCI Design**

   This chapter covers the Human-Computer Interaction (HCI) design, focusing on the development of a user-friendly interface. It presents design principles, early sketches, and the Figma prototype, ensuring the system meets user expectations and provides an intuitive user experience.

5. **System Design**

   The System Architecture chapter explains the design and structure of the system, covering data structures, backend, and frontend design. It details how various components—such as model management and API integration—work together to form a cohesive tool.

6. **Implementation**

   This chapter discusses the Implementation choices, covering key features such that would otherwise not be explained through system design description alone. It also addresses some of the challenges encountered during development and explains how software engineering principles were applied to overcome these issues.

7. **Testing**

   The Testing chapter outlines the methods used to evaluate the system's performance, including unit testing, integration testing, and user acceptance testing. It highlights any issues discovered during testing and describes how these were resolved as well as covering feedback provided.

Conclusion

The conclusion addresses the objectives set out in Chapter 1, evaluating how effectively the project met both its overarching goals and the personal aims defined at the outset. It also includes a critical analysis of the project, identifying potential challenges and limitations encountered during development. Finally, the conclusion outlines potential areas for future work, where further enhancements or expansions could be made to improve the project.

Chapter 2

Requirements

The initial phase of the project centred on gathering and analysing requirements to ensure the system aligned with both the stakeholder's needs and the overall project goals. As this was a team effort, each member had a distinct project brief. A group analysis of these briefs helped to form a unified project brief and problem statement, which were further refined through discussions with the stakeholders.

These meetings allowed for iteration and clarification, during which each team member also refined their individual project requirements based on specific stakeholder feedback. This process led to the final project scope outlined in Chapter 1.

This chapter will explore the requirements-gathering process for the project, detailing the individual project brief, stakeholder discussions, and how these shaped the overall direction of the AI Islands tool.

2.1 Individual Project Brief

"Create a Local Offline Gen-AI with IBM with chatbot characteristics, indexed with all available open-source gen-AI models that can be used off-line and enable selection of different AI models, for testing, benchmarking and passing through to systems integration points. APIs are needed to be designed with enterprise design patterns in mind for MLOps. Provision this as an extension for WatsonX under the IBM-UCL AI Islands name."

This individual project brief presented several initial challenges. While it introduced key concepts and themes, it lacked clarity regarding the stakeholder's specific needs and did not fully articulate the overarching goals of the AI Islands project. To streamline discussions during initial stakeholder meetings, two primary interpretations of the brief were considered:

1. Produce an offline Generative AI chatbot aligned with IBM's chatbot characteristics, equipped with comprehensive knowledge of all currently available offline, open-source Generative AI models. This system would help users select the most appropriate model for their use case and offer testing and benchmarking functionalities to evaluate model performance. The system would also include APIs for external integration.

2. Develop an offline index featuring all available open-source Generative AI models, allowing users to create chatbots with IBM-like characteristics. The models should be downloadable and capable of running offline, with the solution enabling testing, benchmarking, and integration with external services via APIs.

This structured approach to interpreting the brief enabled clearer discussions with team members during the development of an overall Team Brief based on each members individual briefs. This ensured all potential avenues were explored before reaching a consensus. Each interpretation posed distinct requirements, and therefore feedback from the stakeholders was crucial to confirm the most accurate direction before progressing further.

2.2 Stakeholder Discussions

The initial discussions with the stakeholders clarified the project's direction, confirming that the second interpretation of the brief was the intention. The aim was to develop an offline index of open-source Generative AI models, capable of running offline and integrating with external services via APIs. This confirmed direction allowed the team to create a general initial team project brief, incorporating the individual briefs from all team members:

"This project aims to develop a locally-run, offline application that allows users to select from a wide range of open-source AI models for various tasks, including natural language processing, chatbots, visual processing, and audio processing. The application will feature comprehensive benchmarking, testing facilities, and system analytics, enabling users to quickly evaluate model performance for specific use cases.

The app will closely align with IBM's Watsonx online functionality, offering a seamless transition from offline to online for more demanding tasks. It is designed for university students and small businesses that may not be able to afford IBM Cloud costs, while introducing them to Watsonx functionality. APIs will be provided to allow seamless integration with external applications."

In subsequent discussions, this initial brief was refined, and the prioritisation of features became clearer. While benchmarking and testing functionalities were considered essential, it was decided that these would be accessible indirectly through API integration with external applications, allowing users to use their own test scenarios. Additionally, the need for an inbuilt inference UI was emphasised, enabling users to interact directly with the models and review outputs more easily. Features such as performance-based load balancing were deemed outside the scope of the project and considered for future development. Integration with Watson online services was limited to providing an alternative to the offline models in cases where more computationally intensive operations were required, ensuring that the offline index would at least mirror the services available through Watson.

Another significant feature identified during discussions was the introduction of a Playground functionality, allowing users to chain models together and create custom workflows. This would enable users to transition seamlessly between offline and online models, manually selecting the most appropriate model for each stage of a task within the same workflow chain.

The stakeholders also expressed interest in exploring the integration of Retrieval-Augmented Generation (RAG) for chatbots. This feature would allow users to query specific datasets with generative AI models, enhancing the chatbot's ability to provide context-aware responses based on tailored data sources.

Another key takeaway from these discussions was the stakeholder's emphasis on the need for a user-friendly interface, given the potentially non-technical background of the target users.

After these iterative discussions, the individual project briefs were updated and consolidated to form a finalised general project brief, providing clear guidance for the development phase.

2.3 Finalised Team Project Brief

"The aim of the AI Islands project is to develop an offline, locally-run AI application that provides users with access to a wide selection of open-source AI models for tasks such as natural language processing (NLP), Gen-AI chatbots, visual processing, and audio processing. The application will feature an AI model index, allowing users to browse, select, and download models for offline use. Each model will include detailed descriptions, enabling users to choose the most appropriate models for their specific needs.

A user-friendly interface will be developed to ensure accessibility for both technical and non-technical users, including university students and small businesses. The interface will guide users through the processes of model selection, usage, and evaluation, ensuring that even those with limited technical expertise can effectively interact with the AI models. Additionally, an inference section will be included, allowing users to directly test the models they have loaded and review outputs in real time.

The application will also include a Playground feature, enabling users to combine offline and online models into custom workflows. This functionality allows users to create tailored solutions by integrating different models into a single chain, with the ability to switch between locally-run models and IBM Watsonx online services as required. This flexibility ensures that users can scale their workflows according to their specific computational needs.

To support external system integration, the application will offer APIs that enable users to connect the AI models to their own applications, facilitating testing and benchmarking based on their unique requirements. These APIs will provide the flexibility needed for users to embed AI models into existing systems and evaluate their performance within their specific environments.

In alignment with IBM Watsonx, the application will ensure that the offline model index mirrors the functionality of Watsonx's online services, offering consistency when switching between offline and online modes."

The finalised team project brief provided a solid foundation for both general and individual requirement gathering and analysis. It played an important role in guiding the team by clearly defining shared responsibilities as well as contextualising each members individual responsibilities, ensuring all members had a clear understanding of their tasks. Furthermore, it served as a valuable starting point for more detailed analyses, such as the development of personas, use cases, and the MoSCoW requirement prioritisation. By establishing common goals and features, the brief facilitated smoother team collaboration and alignment with stakeholder expectations.

2.4 Personas

During discussions with the stakeholders, several potential user types were identified. Using the finalised team project brief, these were developed into detailed personas, each representing a key user group. This section introduces three distinct personas, illustrating how each has a specific need that the AI Islands application can meet. These personas ensure the project remains focused on delivering solutions that are both relevant and accessible to its target users.

2.4.1 Mark: University Student

Mark, Figure C.1, is a highly motivated undergraduate student who seeks hands-on AI experience but faces budget and time constraints. The AI Islands project will be ideally suited to meet his needs. Its free, open-source platform will provide Mark with access to a wide range of AI models without the financial burden of paid services. The comprehensive model index will allow him to quickly browse, select, and download models that are compatible with his moderate hardware, ensuring he can incorporate AI into his university project without costly infrastructure.

The application's user-friendly interface will simplify the process of selecting and testing models, making it accessible to users with limited technical expertise like Mark. The inference section will enable him to interact with models in real time, helping him evaluate their suitability for his use cases. Additionally, the Playground feature, which will allow for custom model chaining, will give Mark the flexibility to experiment with different models and workflows, deepening his understanding of how AI models function in practice. This will ensure that Mark can efficiently manage his time while gaining valuable hands-on experience.

2.4.2 Alan: Small Business Owner

Alan, Figure C.2, is a small law firm owner who seeks to implement AI to improve efficiency in legal research and document management but has concerns about data privacy. The AI Islands project will provide Alan with a secure, in-house AI solution, allowing him to run AI models locally without the need to transmit sensitive client data to external servers. The offline functionality of the application will directly address his concerns about privacy while offering a cost-effective solution that aligns with his budget.

Through the user-friendly interface, Alan will be able to navigate the model index and select AI models suited to tasks such as document searching and processing, despite his limited technical expertise. The Playground feature will also allow him to create custom workflows by chaining models together, enabling a tailored AI solution that integrates seamlessly with his firm's internal data. This flexibility ensures that Alan can adopt modern technology to remain competitive while safeguarding his clients' confidential information.

2.4.3 Sophie: Teacher

Sophie, Figure C.3, is a secondary school IT teacher who wants to introduce her students to AI in an interactive and approachable way, without overwhelming them with complex tools. The AI Islands project will provide Sophie with a free, offline application that allows her students to experiment with AI models in a hands-on manner, without requiring extensive technical knowledge. The accessible interface will guide students through model selection and testing, helping them gain confidence in working with AI while keeping the learning process straightforward.

The flexibility of the Playground feature will enable Sophie to create custom workflows for her students, allowing them to experiment with different models and configurations suited to various tasks. Additionally, the tool's offline functionality and affordability will ensure that Sophie can effectively integrate AI into her lessons, without exceeding the school's budget or relying on costly cloud services. This makes the AI Islands project an ideal solution for classroom use, offering a practical and engaging way to teach AI concepts.

2.5 Use Cases

Building on the personas defined earlier, this section outlines the primary ways users interact with the AI Islands system. The use case diagram provides a high-level visualisations of interaction

while the use case listing provides a reference for each established use case.

2.5.1 Use Case Diagram

The use case diagram, Figure 2.1, visually represents the core functions of the AI Islands system based on the use case list. It shows how users will engage with the model index, model management, and the Playground feature. It reflects the journey users like Mark, Alan, and Sophie take when selecting, saving, configuring, and running AI models.

GetWatsonData has been highlighted as although it doesn't fall under the personal contributions outlined in this report, it provides contextual information as to the general system interactions.

The use of `<<Extends>>` in the use case diagram indicates an optional or conditional relationship where one use case adds additional behavior to another if certain conditions are met.

Whereas `<<Includes>>` represents a mandatory relationship where one use case explicitly calls another, meaning the behavior of the included use case always occurs as part of the main use case.

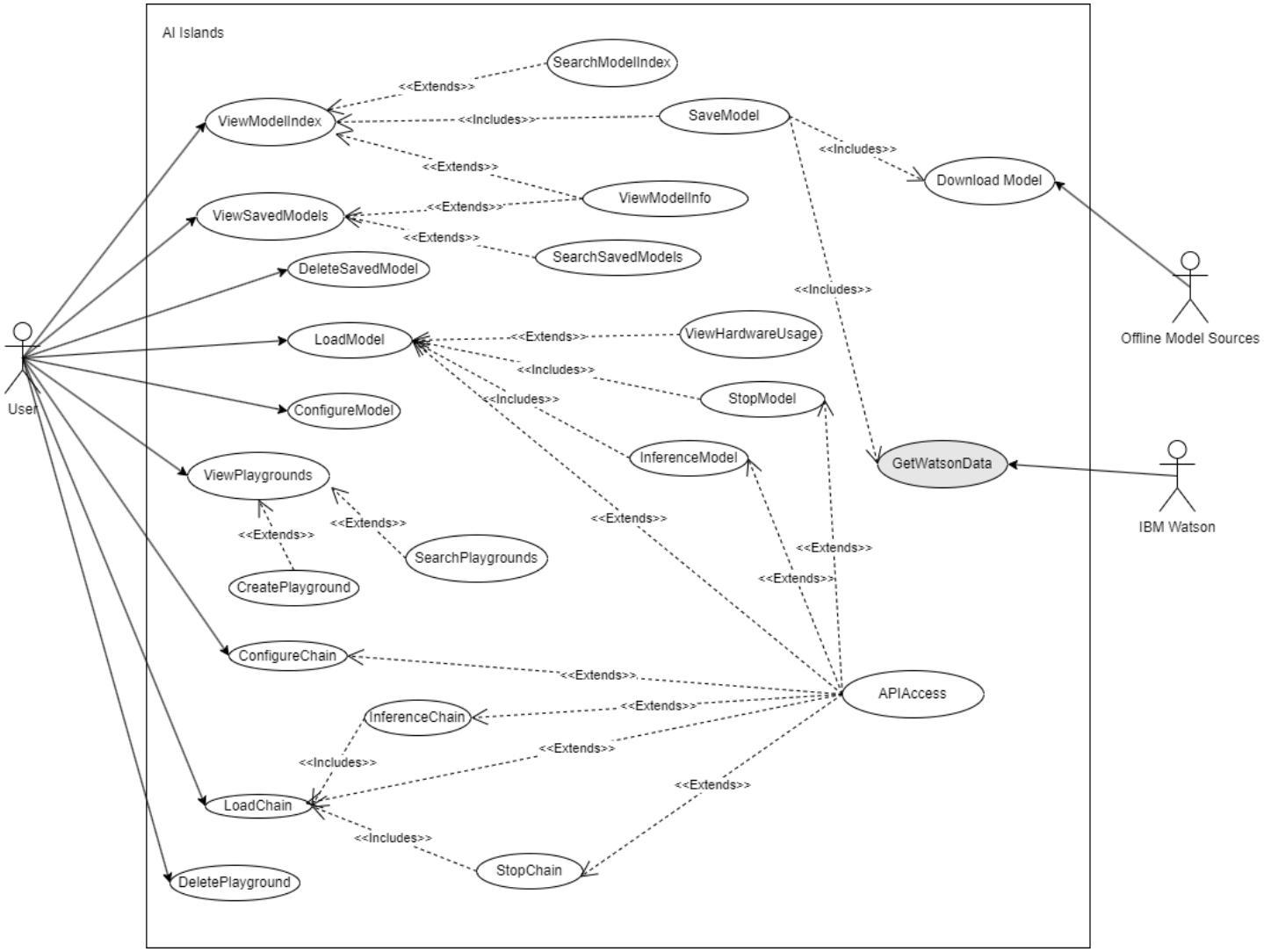


Figure 2.1: Use Case Diagram: GetWatsonData highlighted to indicate other members work.

2.5.2 Use Case Listing

The Table C.1provides a list of each use case identified during the earlier research phases as visualised in Figure 2.1. It features brief descriptions of each use case, providing supporting detail

to the use case diagram.

2.6 MOSCOW Requirements

The MoSCoW method (Must Have, Should Have, Could Have, Won't Have) was used to prioritise the requirements of the AI Islands project. These requirements were informed by the finalised team project brief, Individual brief, personas, and use case analysis. The project brief established the core system goals, the personas identified user needs, and the use cases detailed how users would interact with the system. Together, these elements guided the prioritisation of both team and individual require-ments, ensuring the application addresses the varied needs of its users.

2.6.1 Team Functional Requirements

The Team Functional Requirements define the general requirements necessary to implement the AI Islands application, ensuring that the system aligns with the collective project goals. As team leader, a significant contribution was made in gathering, refining, and establishing these requirements to ensure they aligned with the broader objectives of the project. Their inclusion in this report highlights this contribution, while also providing a clear contextual foundation for the Individual Requirements outlined in subsection 2.6.2. Many of the individual requirements are realised through the specific implementation of these team requirements, demonstrating how they interconnect to achieve both personal and team goals.

The Team Functional Requirements are located in Table C.2.

2.6.2 Individual Requirements

The individual requirements are specifically tailored to meet the objectives outlined in the individual project brief, Section 2.1. While the team functional requirements focus on the application as a whole, the individual requirements ensure the success of specific personal project goals for the integration of Generative AI models for text generation.

This section ensures that the distinct features necessary for achieving the personal project objectives are clearly defined and implemented, independent of the overall system's general requirements.

The Individual Requirements are outlined in Table C.3.

2.6.3 Non-Functional Requirements

The non-functional requirements ensure that the AI Islands application is user-friendly, efficient, secure, and maintainable. These requirements address areas such as interface usability, system performance, data privacy, and error handling. They guarantee that the application delivers a reliable user experience while remaining adaptable and easy to maintain over time.

The Non-Functional Requirements are presented in Table C.4

Chapter 3

Research

Extensive research was necessary to fully understand the technical needs of this project in relation to the brief, stakeholder discussions and gathered requirements. The research began with a broad investigation of similar applications and gradually narrowed in on the specific requirements of this project to determine the models needed to achieve the individual project goals.

Once a solid understanding of the initial technical requirements was established, the focus shifted to backend technology and, finally, to frontend technology.

3.1 Similar Application Review

From the initial project brief and discussions with the stakeholders, it became clear that there were several key features that could function as standalone applications. Due to the extensive nature of the brief, the application review concentrated on applications that demonstrated these key features independently.

3.1.1 AI Model Index: Hugging Face

Hugging Face [2] is a leading platform for hosting and sharing machine learning models, particularly in the field of natural language processing (NLP). It features an extensive index of nearly one million models, including those for vision, audio, text-generation and other domains, which can be viewed in Figure C.19. The platform offers many search and filtering options, such as task type and model name, which are highly desirable features for this project. Additionally, Hugging Face provides tools like the Transformers library and Hugging Face Hub, which allow for easy integration and deployment of models.

While it is primarily an online resource, Hugging Face also offers hosting services that allow users to deploy models directly to the cloud. However, additional steps, such as installing the appropriate libraries and dependencies, are required to deploy models locally, which may present a barrier to entry for inexperienced users.

3.1.2 AI Model loading and interacting: Gradio (with hugging face)

Gradio [3] is an open-source Python library that allows users to create customisable user interfaces (UIs) for machine learning models and other Python functions with minimal coding. It simplifies the process of interacting with models by providing a web-based interface that can be shared with others, making it easy to demo models, collect feedback, or deploy simple applications. When combined with Hugging Face library models, Gradio allows users to quickly develop UIs served in the browser, as can be seen in Figure C.20. The Hugging Face library also offers extensive configuration options for models. The type of model interaction interface provided by Gradio aligns well with the project requirements. However, the need to develop code for implementation and the

complexity of configuration make it somewhat inaccessible to inexperienced users.

3.1.3 AI Model Configuration: OpenAI Playground

The OpenAI Playground [4] is a web-based platform that allows users to experiment with language models by providing custom parameter settings and system prompts, as can be seen on the right hand side of Figure C.21. This implementation enables real-time customisation and experimentation, which are exactly the types of features desired in this project. The platform also offers API integration, allowing users to incorporate their customised models into specific applications. However, the OpenAI Playground primarily provides access to models from the GPT series (e.g., GPT-3, Codex), limiting the diversity of models available compared to platforms like Hugging Face. This makes it an excellent tool for experimentation but less versatile in terms of model selection.

3.1.4 Model Chaining: Kubeflow

Kubeflow [5] is an open-source platform designed to simplify the deployment, management, and scaling of machine learning (ML) workflows on Kubernetes. It allows for the development of highly customisable workflows using Kubeflow pipelines, which can be seen in Figure C.22. Machine learning and AI models can be deployed to separate Docker containers and act as nodes within a workflow. By leveraging containers, Kubeflow provides extensive hardware and performance monitoring, which are highly desirable features for the intended application. However, the barrier to entry for understanding this technology is fairly steep. It does not include a model index, and any models used must be manually added. Although Kubeflow offers a powerful and highly customisable approach, it does not provide the simplicity needed to engage inexperienced users.

3.2 Text-Generation Models

The individual project brief and discussions with the stakeholders necessitated the implementation of several offline open-source generative AI models with chatbot capabilities. The requirement for generative AI significantly narrows the model selection to those based on the transformers architecture, specifically models capable of text generation. While models such as BERT also use the transformers architecture, they are optimised for understanding tasks, such as classification and question answering, rather than text generation. Although BERT can be employed to create chatbots that interpret user intent and extract information, it does not generate responses, and therefore does not fall under the category of generative AI. Nevertheless, its relatively small size and minimal hardware requirements make it an attractive option for users with limited processing power. As a result, the focus of the model selection will be on models designed exclusively for text generation.

3.3 Model sourcing

There are several options for sourcing Text-Generation models, including Hugging Face Model Hub [2], GitHub [6], TensorFlow Hub [7], and PyTorch Hub [8]. Among these, Hugging Face stands out as the most suitable platform for this project due to its extensive model index, robust community support, and its wide range of libraries and tools, including the highly regarded Transformers library and Pipelines. Hugging Face has become a leading platform in the AI and machine learning community, offering a comprehensive ecosystem for discovering, sharing, and deploying models, particularly in the fields of natural language processing (NLP), computer vision, and audio processing.

The transformers library from Hugging Face is a powerful and versatile tool that provides easy access to pre-trained models across a variety of tasks, including text generation, classification, ques-

tion answering, and more. This library simplifies model integration by offering a unified interface to access different models, making it easier to switch between them as needed. Additionally, it supports models from multiple architectures, such as BERT, GPT, and LLaMA, allowing for flexibility depending on the project's needs.

Moreover, Hugging Face's Pipelines offer an out-of-the-box solution for many common tasks like text generation, translation, and summarisation. These pipelines are designed to abstract away much of the complexity involved in interacting with machine learning models, enabling quick setup and implementation. For example, with just a few lines of code, a user can load a pre-trained model and use it to generate text without needing to understand the underlying details of model architecture or training. This ease of use is critical for fast prototyping and deployment, making Hugging Face an ideal choice for projects that require rapid development.

Other model sources, such as GitHub, lack a robust model index and easy-to-use libraries specifically tailored for model deployment, making them less suitable for rapid prototyping. TensorFlow Hub focuses more on TensorFlow-based frameworks, which are less flexible when integrating a diverse range of models, particularly for generative AI tasks. PyTorch Hub offers strong support for PyTorch models, but its collection is not as extensive for pre-trained models or community-driven enhancements. Finally, many other sources often require complex setup processes or lack pre-built solutions for common tasks, increasing overheads for quick model deployment.

Having already reviewed Hugging Face during the similar application review, the platform's extensive offerings and ease of use were well understood. The ability to filter models by task type (e.g., "text-generation") and sort by popularity or performance further reinforces its suitability, as it helps quickly identify the most appropriate models for the project's specific needs.

Filtering the Hugging Face index by the "text-generation" task type and sorting by the number of downloads revealed many potential candidates. While base models often perform adequately, models fine-tuned for specific tasks offer better performance. In particular, models labelled as "instruct" are optimised for handling user instructions and generating coherent responses, making them especially useful for chatbot and conversational AI applications.

3.4 Model Selection

During discussions with the stakeholders, it became clear that model size and system memory usage were critical considerations, with a particular emphasis on keeping memory usage as low as feasible. Given this requirement, model selection must be carefully assessed, as many modern text-generation models have extensive hardware demands.

A trade-off must be considered: models with larger training parameter sizes can capture more complex patterns in their training data, resulting in superior text generation performance. These models often produce more coherent, contextually relevant, and sophisticated responses. However, as parameter size increases, so do the hardware requirements. For example, the Meta Llama 3.1 model with 405 billion parameters requires approximately 810GB of GPU memory to run at FP16 precision compute [9], making it impractical for users with limited hardware and budget constraints. Consequently, the selected models must have a reduced parameter size. It is important to note that while model performance generally scales sub-linearly with parameter size, this is a general observation and might vary depending on specific models and tasks [10]. Nonetheless, the trade-off in performance may not be too prohibitive for basic reasoning tasks.

The focus will therefore be further refined to models with less than 10 billion parameters. For instance, the Meta Llama 3.1 8 billion parameter model requires approximately 16GB of GPU memory using FP16 precision compute. This is much more feasible, as modern high-end consumer

GPUs, such as the Nvidia RTX 3090 and RTX 4090, offer upwards of 20GB of video memory.

A selection of models from the Hugging Face library were chosen, representing a range of parameter sizes within the established 10b limit:

- meta-llama/meta-Llama-3.1-8B-Instruct
- meta-llama/meta-Llama-3-8B-Instruct
- google/gemma-2-2b-it
- google/gemma-2-9b-it
- microsoft/Phi-3.5-mini-instruct
- Qwen/Qwen2-0.5B-Instruct
- Qwen/Qwen2-1.5B-Instruct
- mistralai/Mistral-7B-Instruct-v0.3

This selection represents a subset of the most popular open-source text-generation models fine-tuned for instruction-based tasks and will be used as a basis for creating the offline index described in both the team and individual project briefs.

3.5 Backend Technology

3.5.1 Language choice: Python

Python is a high-level, versatile programming language that has become the dominant choice for machine learning (ML) and artificial intelligence (AI) development due to its simplicity, extensive libraries, and strong community support.

Python's ecosystem includes powerful libraries and frameworks such as TensorFlow, PyTorch, scikit-learn, and Keras, which simplify the building, training, and deployment of machine learning models. These libraries abstract many of the complexities involved in ML development, allowing developers to focus on the core logic and design of AI systems rather than on low-level implementation details. Furthermore, libraries like NumPy and Pandas are widely used for data manipulation and analysis, which are essential components in ML and AI workflows.

The project requires the ability to integrate models from various sources and task types, and Python is well-suited for this. It offers broad support for handling different models and frameworks, providing compatibility with a wide range of model sources. Python's flexibility also allows for the integration of multiple tools and libraries, making it a good choice for expanding the project's scope as required.

Another key consideration is the fact that the Transformers library, a critical component for model implementation in this project, supports only Python and JavaScript. While JavaScript presents a powerful option, it is more appropriate for web based applications and given the requirement for a desktop implementation, Python was given preference. Python is also the more popular choice for backend development in ML and AI applications, this further solidified the decision to use Python as the backend language for the project.

With the programming language and model source selected, development began on prototype applications to test implementation possibilities. The development process followed a staged approach according to identified features in the use cases: download, load, inference, and configuration, with each step building upon the previous. These prototypes allowed for a thorough exploration of the capabilities of the Transformers library and Pipelines package.

The decision to use Pipelines was made early in the prototyping phase, as it abstracted many complexities related to handling diverse models for different task types. This abstraction reduced the need for manual imports, which would have otherwise posed a challenge for an application requiring dynamic handling of multiple models.

The following sub sections will discuss key conclusions and findings from the prototyping and research phases, addressing both the specific project brief and the broader AI Islands application requirements.

3.5.2 Transformer Library

The decision to use the Transformers library led to research into how model downloading, loading, inference, and configuration should be handled.

3.5.2.1 Model Downloading

Three main methods were identified for downloading models from the Hugging Face index.

1. **Using the huggingface_hub library**:
   The hf_hub_download module provides a simple, highly controlled approach. However, this method often results in larger download sizes, likely due to downloading files not directly related to the model loading process.

Listing 3.1: Python pseduo code for download using huggingface_hub

```
from huggingface_hub import hf_hub_download

model_file = hf_hub_download(repo_id="model name", cache_dir="download path")
```

2. **Pipelines API**:
   This method reduces the process of downloading and loading a model to simply specifying a model name from the Hugging Face index and a task type. While straightforward, it lacks the flexibility to select custom tokenizers, which is essential if a user needs to define a custom tokenizer for a model.

Listing 3.2: Python pseduo code for download using Pipelines API

```
from transformers import pipeline

generator = pipeline("task type", model="Model Name", cache_dir="download path")
```

3. **Using AutoModelForCausalLM and AutoTokenizer modules**:
   Although more complex, this approach ensures only the necessary files are downloaded, reducing storage requirements, especially important for large language models. It also allows for future expandability, as users can replace AutoTokenizer with a custom tokenizer for specific configurations.

Listing 3.3: Python pseduo code for download using Auto classes

```
from transformers import AutoTokenizer, AutoModelForCausalLM

tokenizer = AutoTokenizer.from_pretrained("gpt2", cache_dir="Download Path")

model = AutoModelForCausalLM.from_pretrained("Model Name", cache_dir="Download Path")
```

Tokenizers are essential for converting raw text into tokens since machine learning models cannot process text directly. Tokenizers break the text into smaller units and encode it into numerical representations. They are model-specific because each task and model type requires different pre-

processing before inference. The AutoTokenizer module automatically detects and downloads the appropriate tokenizer from the model repository, simplifying model-specific tokenisation.

Causal Language Modelling (Causal LM) predicts the next token in a sequence based on the preceding tokens. This approach is particularly relevant to text-generation models, where the next word is generated based on the prior input.

The third option, using AutoModelForCausalLM and AutoTokenizer, proved to be the most effective. It reduced storage requirements, critical for large language models, and allowed for immediate testing of whether a model could be loaded by a specific system. Unlike the first method, this approach loads the tokenizer and model directly for inference after downloading.

3.5.2.2 Loading, Inference, and Configuration

When loading cached models, two primary methods were considered: using the auto modules or the pipelines module. It was found that certain configurations, such as device type and optimisation, needed to be applied directly to the model and tokenizer during instantiation. These configurations could not be passed as parameters in the pipelines method with optimisations implemented.

The auto classes also allow for greater flexibility, especially if fine-tuning is required. However, pipelines simplify model inference, handling tokenisation, padding/truncation, and input management automatically, meaning only the input text and optional pipeline parameters are required.

A combination of both methods was therefore selected to balance configuration flexibility with ease of use. The model and tokenizer are first loaded and then passed as variables to the pipeline in order to leverage its simplicity of use.

Listing 3.4: Python pseduo code for downloading, loading and inferencing.

```
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline

tokenizer = AutoTokenizer.from_pretrained("Model Name", cache_dir="Download Path", args)

model = AutoModelForCausalLM.from_pretrained("Model Name", cache_dir="Download Path", args)

text_generator = pipeline("Task Type", model=model, tokenizer=tokenizer)

output = text_generator("Input text", pipeline_args)
```

This method provided an ideal balance between configuration flexibility and ease of inference. It directly addresses NFR-6, ensuring maintainability and future expandability.

3.5.3 FastAPI

One of the key requirements of the AI Islands project is the ability to serve API requests, enabling end users to integrate AI models into their own use cases once configured and customised. This allows users to benchmark and test multiple models against their specific requirements without the need to download and integrate each model manually.

The need for a robust REST API framework becomes even more significant when considering frontend technology, as the stakeholders required the use of a different programming language for the frontend. This necessitates that all frontend-to-backend interaction be managed through an appropriate API framework.

There are several Python API frameworks available, including Flask, Django, Tornado, and FastAPI. While all offer robust features, performance and asynchronous capabilities are particularly important for the AI Islands application. Without the ability to handle requests asynchronously, any inference request would block the server from processing other requests until the operation completes.

This would limit throughput, especially if multiple inference requests are made simultaneously, reducing the number of requests that can be processed in a given time.

FastAPI was selected due to its strong native support for asynchronous requests and its optimisation for high performance. While other frameworks would also have worked and, for a project of this scale, the choice is somewhat flexible, FastAPI was considered the best option in light of future scalability.

With the API framework chosen, the prototype applications could now be interacted with using tools such as Postman. This provided a more versatile interface compared to the previous command-line implementations and paved the way for creating a more sophisticated interface.

3.5.4 Multiprocessing

Once the successful loading of AI models from the Transformers library was achieved, attention shifted to accommodating the requirement of having multiple models loaded into system memory concurrently. This was essential for meeting the needs of the playground and facilitating simple model comparisons. While loading models simultaneously did not present significant issues, the challenge arose with unloading them. Standard methods, such as garbage collection, were insufficient in fully removing instances of the models from system memory. This led to large portions of memory being occupied after loading and unloading multiple models, with the only viable solution being to restart the application to free up resources.

Research was therefore conducted into potential solutions to mitigate this issue, and two viable options were identified. The first solution involved using containers, as discussed in the similar application review of Kubeflow. Containerisation would allow models to be loaded and unloaded efficiently, improving scalability as well. However, this approach had a steep learning curve and, given the project's time constraints, would have required significant effort before a functional application could be developed. While this solution would have been ideal for a more scalable setup with potential for server deployment, it was deemed impractical given the project's scope and timeline.

The second option was to load AI models into child processes spawned from a parent process. This approach had a much lower learning curve, as inter-process communication (IPC) had been previously covered in the context of computer architecture and operating systems. Given that scalability was not a primary requirement, this solution was considered the more appropriate choice.

Various Python modules were evaluated for this purpose. Subprocess and Multiprocessing were both investigated, with Multiprocessing ultimately selected due to its high-level API and its ability to bypass Python's Global Interpreter Lock (GIL). The GIL, if not addressed, would have resulted in significant performance bottlenecks when running multiple models concurrently. By leveraging the Multiprocessing module, each model could be loaded into its own process, allowing for true parallelism across CPU cores. This enabled the models to operate independently of one another without being constrained by the GIL.

This approach not only ensured more efficient CPU utilisation but also simplified the unloading of models, as each process could be cleanly terminated, freeing up system memory without requiring a full application restart.

An additional benefit of using the Multiprocessing module was also identified. Since each model operates within its own child process, each with a unique Process ID (PID), hardware and performance monitoring became significantly more straightforward. CPU usage, system memory consumption, GPU usage, and GPU memory usage were all easily extracted and clearly attributable to specific models, as each was contained within its own distinct process. This feature enabled

the fulfilment of a key requirement of the AI Islands application: the ability to monitor the performance of individual models.

This research led to the development of a functional prototype that was capable of downloading, loading, and unloading multiple models from system memory. This prototype served as the foundation for the main application's development and had a significant influence on the class structure, which will be discussed in Chapter 5.

3.5.5 Fine Tuning

Fine-tuning was explored for text-generation models using the Transformers Trainer library. This process involved reading a custom dataset, tokenising it with the previously downloaded tokenizer, defining training parameters such as the number of epochs, and then applying these to an instance of a trainer object.

While the implementation was technically feasible, further investigation revealed that the time required to effectively train a large language model on modest hardware was highly impractical, potentially taking several days.

This posed a significant limitation to development, as each new feature implementation would require extensive testing before being integrated into the application. Additionally, the text-generation models were already fine-tuned for instruction-based tasks, and further fine-tuning risked reducing accuracy for chat-based functionality. As a result, it was decided that fine-tuning would not be included for text-generation models in this initial version of the AI Islands application. Instead, alternative methods for handling user data would be considered, such as RAG.

3.5.6 RAG

Retrieval-Augmented Generation (RAG) enhances text-generation models by enabling them to retrieve and incorporate external data during the generation process. Unlike traditional models, which rely solely on pre-trained knowledge, RAG models can query user-specific or custom datasets in real time, delivering more accurate and contextually relevant outputs. This capability is particularly valuable for tasks that require domain-specific knowledge or personalised responses. By accessing custom data, users can tailor the model's outputs to their specific requirements, making RAG well-suited for tasks like personalised question-answering, targeted content generation, and extracting data-driven insights.

The RAG process begins with an embedding model, which is selected to process the dataset. This model converts the dataset into high-dimensional vectors that capture the semantic meaning of the data. Once processed, the resulting embeddings are stored in an index using a similarity search tool.

When a user queries the dataset, the embedding model encodes the query into a similar high-dimensional vector. This query vector is then compared against the indexed vectors to retrieve those with similar semantic meaning. The original data associated with the retrieved vectors is passed to a large language model along with the user's query, providing context-specific information that enhances the generation process by incorporating relevant external data.

The task, therefore, was to select an appropriate embedding model and similarity search tool. For the purposes of this individual project brief, only the following models were considered, though the final application may include additional options, as other team members were also exploring RAG-based solutions.

Given the already significant computational demands of running a large language model offline, it was essential to prioritise embedding models that were not only efficient but also optimised for

speed. This led to the selection of the all-MiniLM-L6-v2 embedding model. While it does not offer the same level of performance as larger models such as SBERT or T5, it strikes a reasonable balance between size and accuracy. For many use cases, such as document-based question-answering, the retrieval accuracy difference between these models is often negligible.

The key advantage of all-MiniLM-L6-v2 is its minimal memory usage and fast processing, even when deployed on a CPU. This makes it an ideal choice given the substantial hardware requirements of the text-generation models being used, helping to keep the overall system efficient and responsive.

Selecting the right similarity search tool was essential for efficient data retrieval. FAISS (Facebook AI Similarity Search) was chosen for its strong performance in large-scale searches, handling large datasets with a good balance between speed and accuracy. This makes it well-suited for real-time retrieval in a RAG system, where fast querying of custom datasets is key to providing relevant context to the text-generation model. FAISS is optimised for both CPU and GPU environments, offering the flexibility needed for the AI Islands project, where hardware capabilities may vary. This ensures speed and responsiveness even on modest setups, while also allowing the system to scale up for more demanding tasks on GPU hardware. Alternatives like Annoy were considered, but Annoy tends to be less accurate with large datasets and does not scale well on GPUs, making FAISS the more reliable and versatile choice for this project.

3.5.7 Data Structure

In this project, both structured and semi-structured data storage solutions were considered to meet the diverse requirements of handling multiple AI models from various sources. Structured data storage, such as relational databases, offers a well-defined schema that is highly organised and ideal for scenarios where rigid data structures, consistent data types, and transactional integrity are required. However, given the nature of the project, which involves working with a wide range of AI models, each with its own configuration needs, the fixed schema of structured databases would likely become cumbersome and inflexible.

In contrast, semi-structured data storage offers a more flexible approach. Semi-structured data formats, such as JSON, allow for data to be stored without a predefined schema, making it easier to accommodate varying data types and evolving configurations. This flexibility is particularly important in a project that needs to integrate AI models from diverse sources, each with its own distinct set of configuration requirements. While a minimum set of required data fields could be defined, each model source would be able to extend or modify this structure as needed. Attempting to implement this flexibility in a structured database would likely introduce significant complexity, requiring constant schema adjustments to manage heterogeneous data formats.

Given this analysis, JSON emerged as the preferred data format. It provides the adaptability needed for handling models with diverse configurations, while also offering a seamless integration with API-based interactions. As JSON is the standard format for API requests, its use as a primary data structure would improve efficiency by minimising the need for conversions during data exchanges between the frontend and backend.

When considering how to store and manage this semi-structured data, two options were evaluated: direct JSON file interaction and the use of a NoSQL database like MongoDB, which also uses JSON-like document structures. While a NoSQL database could provide similar flexibility in handling varied data, for this project's initial phase, direct JSON file interaction presents certain advantages.

First, direct JSON file interaction would allow for on-the-fly development, enabling rapid iteration

and adaptation during the early stages of the project without the overhead associated with managing a database system. This would be particularly beneficial in an initial version of the application, where features and model requirements may still be evolving. Moreover, the initial scope of the project involves a curated set of AI models, meaning the performance and scalability concerns associated with direct file handling would not pose significant challenges at this stage. By avoiding the need for database setup and management, the architecture could remain lightweight, allowing for a more agile development process.

In addition, the integration between the frontend and backend is simplified by using JSON files directly, as the data format aligns with standard API requests. This would streamline the system's data handling, allowing the development team to focus on core functionality rather than dealing with additional layers of complexity introduced by a database.

3.6 Frontend Technology

3.6.1 Language choice: C#

Several UI frameworks were researched and considered for the AI Islands application. Initially, Python-specific frameworks such as Tkinter, CustomTkinter, PySide2, PyQt5, and Flet were explored due to their potential for easier integration with the backend. While Tkinter and CustomTkinter offered straightforward implementation, they were too basic and did not meet the project's requirements for an intuitive, user-friendly interface. PySide2, PyQt5, and Flet provided greater flexibility in UI design but came with a steeper learning curve and lacked some of the modern design capabilities needed for a polished user experience.

Given the application's need for extensive API integration, this opened up the possibility of exploring non-Python-based frontends, as the backend was designed to communicate through APIs. This led to the consideration of modern web-based frameworks such as Next.js, using JavaScript for the frontend. By leveraging Electron, this approach would have allowed the use of web technologies like HTML and JavaScript to build a desktop application, offering considerable resources and flexibility in UI design. However, this option was ultimately discarded after stakeholder discussions due to concerns over Electron's high memory usage, as it relies on the Chromium browser engine, which is known to be resource-intensive.

During stakeholder meetings, it was determined that the Python-native frameworks were insufficient to meet the desired level of sophistication for the interface. After presenting various UI design options, the stakeholder saw potential for broader adoption of the application by different companies and believed that C# .NET would be the most agreeable option in terms of industry usage. While the team had explored alternatives like Electron with Next.js, the stakeholder felt that C# would best align with the usability and adoption goals, providing better control over memory usage and performance expectations.

As a result, the final decision was made to implement the frontend using C# .NET. This led the team to select .NET MAUI as the UI framework due to its cross-platform support and high performance, which aligned with the stakeholder's long-term vision for the application.

Chapter 4

HCI Design

This Chapter outlines the user interface design, detailing the HCI principles applied and how the UI is structured to meet these criteria. It will also present the initial draft sketches, followed by the development of an interactive Figma prototype.

4.1 Design Principles

Human-Computer Interaction (HCI) Design focuses on creating systems and interfaces that provide efficient, intuitive, and enjoyable interactions with computers. The primary goal is to meet user needs through a user-centred design approach, which emphasises understanding user behaviours and challenges through research and testing.

As discussed in Chapter Two, conversations with the stakeholders emphasised the importance of a user-friendly interface. The subsequent development of personas helped establish a clearer understanding of the potential users of the AI Islands application, each with unique needs and objectives. This user-driven approach to defining design requirements is central to HCI design, reinforcing the importance of creating an interface that is both accessible and intuitive, without sacrificing the more technical features, such as model configuration.

Additionally, the application review in Chapter Three offered key insights into how similar application features could be expressed. These combined processes laid the foundation for the UI design, which will be presented in the following sections.

4.2 Paper Sketches

Paper sketches were use as an initial means to help conceptualise the UI design. They allowed for quick design iteration and collaborative discussion within the team, without the time investment associated with a wire frame design or Figma prototype.

They were also used during stakeholder meetings where project ideas could be more effectively expressed, consolidated and clarified.

They represented an important stage in the design process and also helped provided insight into the necessary requirements of the backend system as a means to serve the established features.

The sketches paved the way for more sophisticated design implementations such as Figma Prototypes. A full list of the sketches including detailed descriptions can be found in Section C.3.1

4.3 Figma Prototype

The Figma prototype was the result of a collaborative effort between the project team members. Each member contributed their own paper sketches, followed by a group review to identify the most

desirable features in line with the established requirements and personas. This prototype represents a key development from the initial paper sketches, where features were added or revised.

A comprehensive visualisation of the Figma Prototype, alongside descriptions with comparative analysis against the Paper Sketches can be found in Section C.3.2.

4.4 Feedback and Changes

The Figma prototype was showcased during a team presentation held at the IBM York Road offices in London on June 28th, 2024. The audience included the project's industry stakeholder, Prof. John McNamara, as well as the internal stakeholder, Prof. Dean Mohamedally, alongside IBM professionals and executives.

The feedback from both industry and internal stakeholders was overwhelmingly positive, offering the confirmation needed to proceed with the implementation of the design.

The Figma prototype served as a solid foundation for UI development, with the overall design themes retained. Only minor adjustments were in the final application.

Chapter 5

System Design

The AI Islands requirements (specifically NFR-6 from Table C.4) call for a maintainable code base with a strong focus on modularity and expandability. To achieve this, an appropriate system design was crucial. Drawing from prior experience in the 'COMP0066: Introductory Programming and particularly the 'COMP0071: Software Engineering' modules, the Entity-Control-Boundary (ECB) design pattern was selected as a suitable architecture for the backend system.

The ECB pattern is an object-oriented design pattern that divides the system into three key categories, enabling modularity, ease of maintenance, and clear system organisation. For this backend system, ECB offers a clean separation of responsibilities, with entities handling business logic, controllers managing the application flow, and boundaries acting as interfaces between external systems and the internal logic.

While other patterns like Model-View-Controller (MVC) or Layered Architecture could have been considered, these designs are more tailored to user interface-driven applications or involve more rigid layering structures. MVC, for instance, focuses on separating UI concerns, which is less relevant to the backend focus of AI Islands. Layered Architecture, while promoting separation, can lead to excessive complexity and tight coupling between layers. ECB, by contrast, allows for a flexible and lightweight structure, making it ideal for a backend system where clarity, maintainability, and the ability to easily integrate or expand components are paramount. This aligns more directly with the project's need for a scalable and modular backend architecture.

- **Entity Classes:** These represent the core data models of the system, encapsulating business logic specific to the entities and managing their data.
- **Control Classes:** These classes handle the business logic that interacts with the entity classes. They process inputs from boundary objects (like user interfaces or API calls) and are responsible for managing the interactions between different entity objects and other control classes.
- **Boundary Classes:** Serving as interaction points between the system and external actors (end users or external systems), these classes manage communication with control classes, effectively bridging the outside world and the system's core logic.

This chapter will walk through the process of implementing this design pattern, with each decision made specifically to align with the project's modularity and maintainability requirements.

5.1 Component Diagram

The component diagram below builds on the principles discussed in the system design introduction. Using a black-box approach, it presents a high-level overview of the AI Islands system, displaying key modular components and their interactions.

The system is divided into two distinct subsystems: the Frontend and the Backend, each featuring subsystem-specific components that communicate via HTTP requests provided by the backend. These HTTP requests are also exposed to the system boundary, meeting the requirement for external application integration through API requests.

Frontend SubSystem
The Frontend is responsible for requesting the necessary UI data, formatting it into a user-friendly interface, and presenting it to the end user. The internal workings of the UserInterface component have been abstracted in this diagram because the exact implementation depends on the chosen frontend technology or framework (e.g., C#, Next.js, etc.), making it interchangeable.

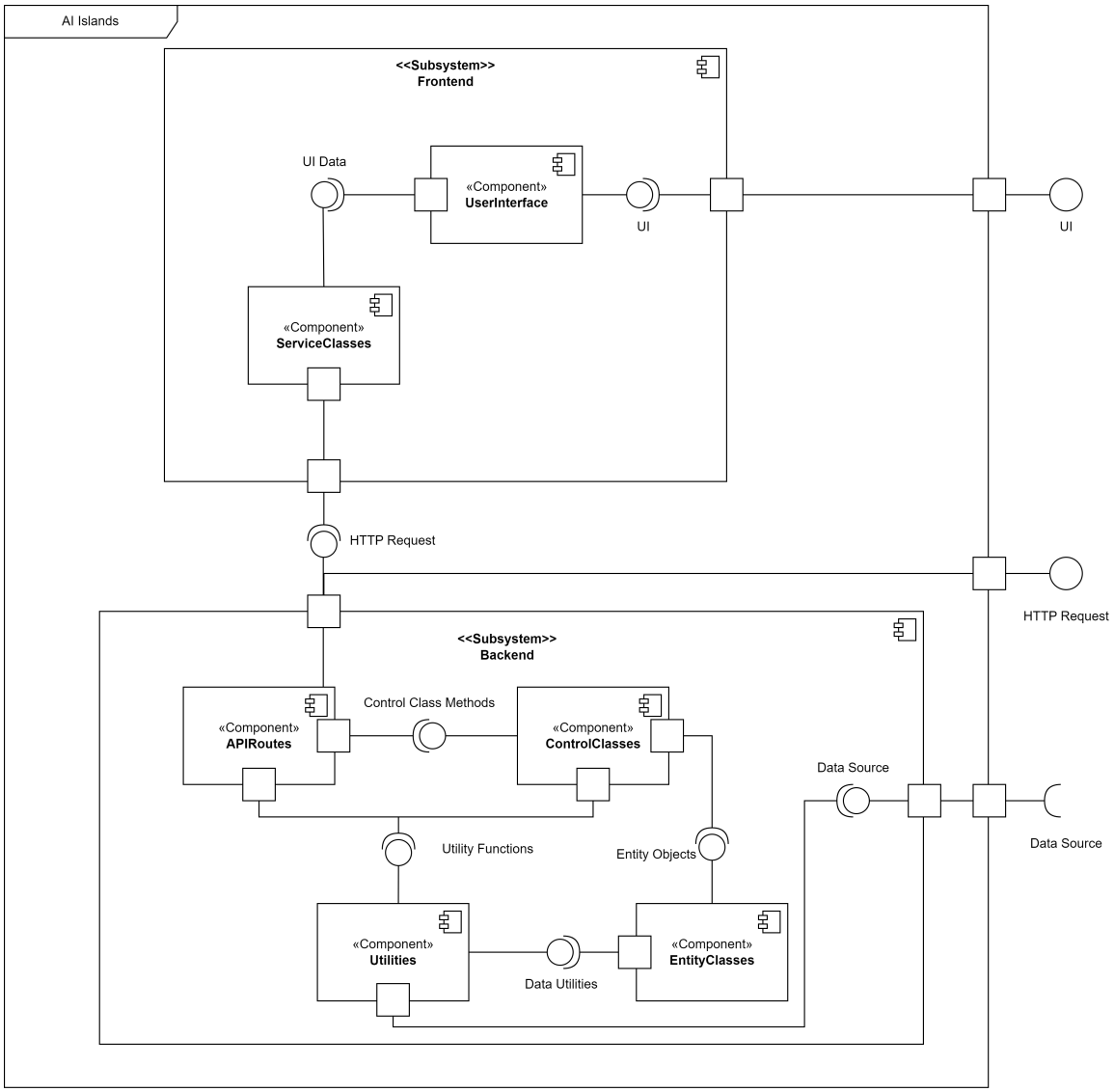


Figure 5.1: Component Diagram

Backend Subsystem
The Backend subsystem adheres closely to the ECB pattern, with clear separation between the boundary, control, and entity layers:

- The APIRoutes component acts as the boundary, interfacing with external requests and facilitating communication between the external environment and the control classes.
- ControlClasses manage the business logic by interacting with the EntityClasses through method calls, ensuring data is processed and managed appropriately.
- The EntityClasses encapsulate data models and related logic, ensuring a separation of concerns between data management and business logic.

Additionally, the Utilities component plays a critical role by providing reusable assets and data access functions, enabling interaction with either local databases or external data sources. This modular approach ensures scalability and ease of maintenance, while also supporting the system's data access requirements.

5.2 High Level Class Structure

The diagram in Figure 5.2 outlines the high level class structure for the AI Islands system, showing key interactions across both the frontend and backend components. While it highlights the major classes, the primary focus is on visualising how these classes interact with one another, rather than detailing class specific functionality. Dotted lines in the diagram represent a general "Uses" relationships, while solid lines indicate an inheritance/extends relationship.
Furthermore stereotypes ,<<Stereotype>>, have been used to identify class type. More detailed explanations of the internal workings and attributes of these classes will be provided in the Backend Design and Frontend Design sections.

In this structure, the system follows the aforementioned ECB pattern. The frontend comprises various service classes such as PlaygroundService, ModelService, and LibraryService, which interact with the UI and communicate with backend routers. These routers (e.g., ModelRouter, PlaygroundRouter) are responsible for directing requests to the appropriate control classes within the backend.

The backend manages core logic and entity handling through control classes like ModelControl and PlaygroundControl, which oversee entities such as AI models (TransformersModel, UltralyticsModel, and WatsonModel), all of which inherit from a base class, BaseModel. This inheritance ensures that common functionality is centralised, allowing the system to be easily extended with new AI models.

It is important to note that while this diagram focuses on class interactions, many interactions with utility classes and the UI have been abstracted for clarity. The UI is abstracted because it follows a different framework and architecture from the rest of the system, and its design will be further elaborated in the Frontend Design section. Similarly, the Utilities class is abstracted because there are many utility functions across the system, and listing them all would be impractical. These utility functions, while important for common system-wide tasks, are more involved in implementation than in system design and have therefore been represented as an abstraction. As a result, both the UI and Utilities components have been labelled with the <<Abstraction>> stereotype in the diagram.

This choice of abstraction ensures that the diagram remains focused on the key system components and interactions, while allowing for deeper discussion of the UI and utility functions in their respective sections.

This diagram provides a reference for understanding the relationships between key components, laying the foundation for the more detailed class specifications that follow.

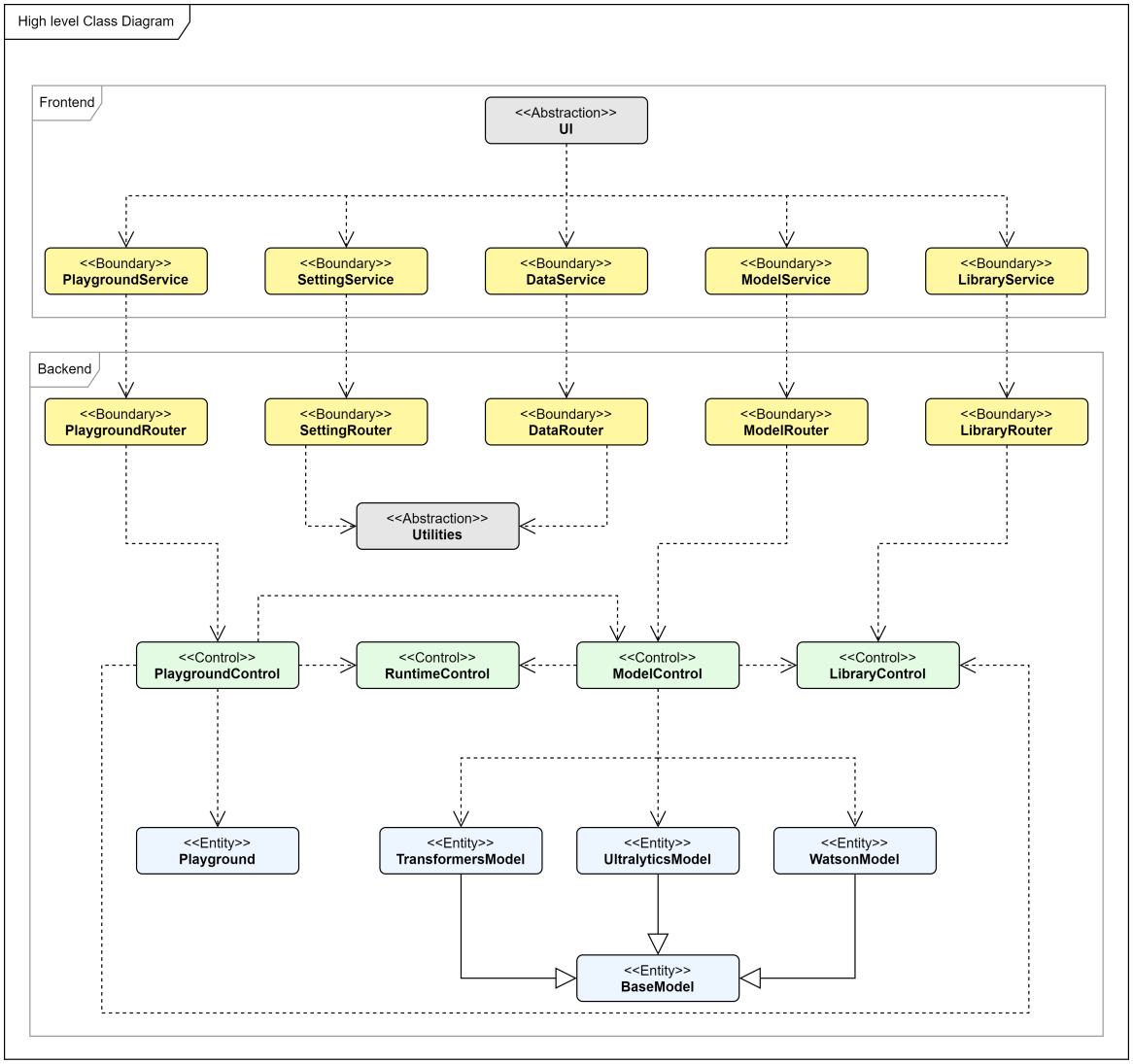


Figure 5.2: High Level Class Diagram

5.3 Backend design

With the design pattern established, the backend design was developed by carefully considering the project's requirement for multiple model and model source integrations, with a strong emphasis on expandability. Three main model sources were identified, and corresponding entity model classes were created: TransformersModel, UltralyticsModel, and WatsonModel. Each of these sources had different instance variable requirements, and it was essential to maintain this flexibility for future expandability, as it is impossible to predict the specific requirements of future model sources.

The focus, therefore, was on defining a minimum set of methods that would provide the necessary functionality, regardless of the model source implementation. These methods were identified as download, load, and inference. This ensures that any future model source must at least imple-

ment these methods to integrate successfully with the application. To represent this requirement, a BaseModel abstract class was created.

With these common methods established, work could begin on creating a ModelControl class that routes access to these methods based on an identifying piece of data, in this case, a string variable named model_id. The control class would then access the relevant model class to trigger the required functionality. As the ModelControl class is responsible for managing interactions with the entity model classes, it required several fundamental methods: load_model, download_model, unload_model, and delete_model. Additionally, it needed an instance variable to store instantiated model objects. This initial class brainstorming can be seen in the Appendix Figure C.23.

To enable data access for populating the entity classes, the ModelControl class required integration with an additional control class, LibraryControl. The LibraryControl class is responsible for accessing, manipulating, and writing index and library data, delegating direct file interactions to utility functions. This separation of concerns ensures that data handling is well-structured and maintainable.

In a similar approach to the model entity-control design, a Playground entity was established, primarily functioning as a data container. It stores information about the models it includes, chain configurations, and descriptive metadata. To manage playground-related functionality, the PlaygroundControl class was created. This class is responsible for instantiating and managing Playground entities, overseeing all interactions within the playground environment. For any operations involving models in the playground, PlaygroundControl specifically utilises ModelControl to handle model-related tasks, ensuring a clear separation between playground functionality and model management.

During implementation, the need for a dedicated control class to manage runtime data became evident, leading to the creation of the RuntimeControl class. This class is used by both PlaygroundControl and ModelControl to read and write runtime data, with utility functions facilitating direct interaction with data files.

By adopting a bottom-up development approach, starting with entity classes and moving to control classes, the design process enabled a clearer understanding of how to segment the system's external-facing boundary classes. Each boundary class was categorised according to the entity or data it interacts with. For instance, the PlaygroundRouter handles interactions related to the playground via PlaygroundControl, while the SettingRouter and DataRouter interface directly with utility functions to manage system-wide functionality. The ModelRouter manages model-specific functionality by working with ModelControl, and the LibraryRouter oversees index and library functionality through LibraryControl.

Each boundary router contains a carefully defined set of functions that provide the necessary capabilities for interacting with the system, ensuring a clean and efficient separation of concerns throughout the backend design, with a clear delineation of responsibilities that promotes high levels of code maintainability.

5.4 Backend Class Structure

The class diagram shown in Figure C.24, offers a detailed view of the backend system, focusing specifically on the contributions outlined in this project report. Certain classes, such as WatsonModel and UltralyticsModel, have been excluded as they were implemented by other team mem-

bers.

The diagram uses different types of arrows to represent various relationships:

- **Dashed line arrows** signify a dependency relationship, where a class relies on another class's functionality for short-term operations.
- **Solid line arrows** indicate an association relationship, where one class maintains a long-term reference to another to provide class-specific functionality.
- **Solid lines with hollow arrows** represent an extends relationship, where one class inherits from another class.

These detailed relationships provide additional context on how the system components interact with one another.

For simplicity, the SettingsRouter and DataRouter have been omitted from the diagram, as their functionality is limited to interactions with utility classes. Instead, the focus is on illustrating the implementation of the ECB (Entity-Control-Boundary) design pattern. With this in mind, a single utility class, JsonHandler, is included. This utility class is integral to the system's design, acting as a boundary utility responsible for reading and writing data files.

This diagram represents the final product of an iterative development process. Each method and variable was carefully evaluated as new features were implemented, building upon the initial structure described earlier in the report. For details on the implementation of some of these methods and classes, please read Chapter 6: Implementation.

5.5 Sequence Diagrams

To provide a clearer visualisation of how different classes interact through method calls, three key system interactions have been depicted using sequence diagrams. These diagrams complement the class relationships outlined in Figure C.24, offering additional context on how the backend components of the AI Islands system work together.

The sequence diagrams present a high-level overview of class interactions within the backend, without delving into the specific implementation details of individual methods. They focus on illustrating how methods from one class invoke operations in other classes. The lifelines represent when classes are instantiated, while the activity bars show the period during which a particular object or lifeline is actively performing an operation or processing a request.

5.5.1 Sequence 1: Downloading A TransformerModel

Sequence 1 (Figure 5.3) illustrates the system interactions that occur when a client (either via the UI or through an API call) requests the download of an AI model from the Transformers library. This is particularly relevant to the individual project brief, as it demonstrates the process by which a client initiates the download of a generative AI text-generation model.

While it is generally uncommon to include specific implementation details in sequence diagrams, an exception has been made here to clarify a key feature of the system's design. In this sequence, the ModelControl class features a secondary activity bar, highlighted in red, which represents op-

erations executed in a separate process. This detail has been included because it directly influences how the TransformerModel lifeline is represented in the diagram.

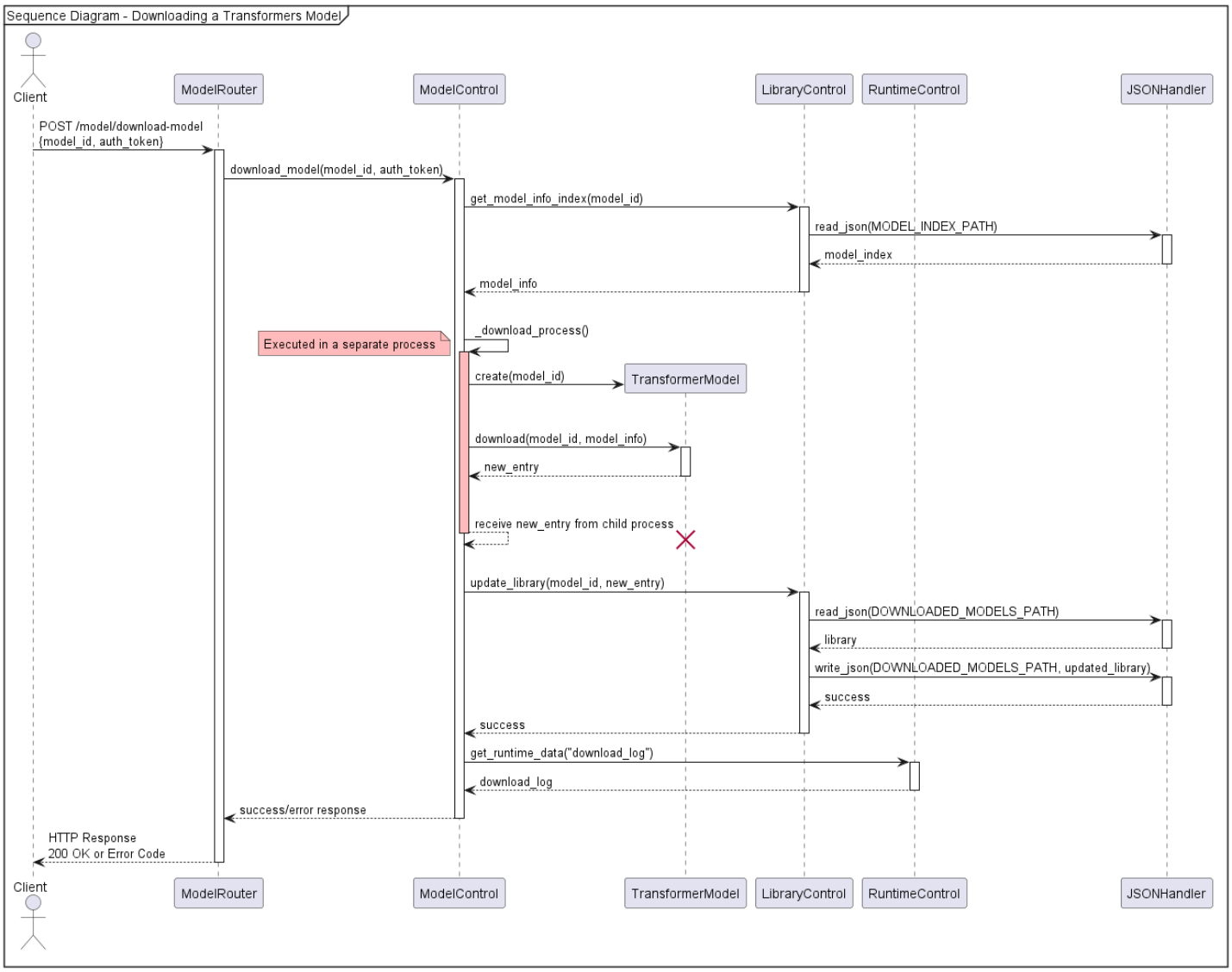


Figure 5.3: Downloading Transformers Model

The sequence begins with the ModelRouter handling a request from the client to download the model. This request is passed to the ModelControl, which retrieves model information from the LibraryControl via a call to get_model_info_index(). The download process then proceeds with the creation of a new instance of the TransformerModel, handled within a separate process (depicted by the red activity bar in the diagram).

It is important to note that the TransformerModel lifeline begins at the point where the separate process is initiated by ModelControl. This is why its lifeline is shorter compared to other classes, as it only becomes active once the model is instantiated within the separate process. The model creation and download are executed within this child process, after which the results (a new model entry) are passed back to the main process.

Upon completion, the child process terminates, as shown by the red cross at the end of the TransformerModel lifeline. At this point, the model instance is destroyed along with the child process, as the TransformerModel is not required beyond the download operation. Following this, Model-

Control updates the library data by writing the new model information to the downloaded models file and proceeds to return a success or error response to the client, marking the end of the sequence.

Including this detail in the sequence diagram helps to clarify how the system manages key interactions during the model download process. This specific approach to handling model interactions will also be represented in subsequent diagrams, providing consistency in understanding the system's class interactions.

5.5.2 Sequence 2: Loading A Playground Chain

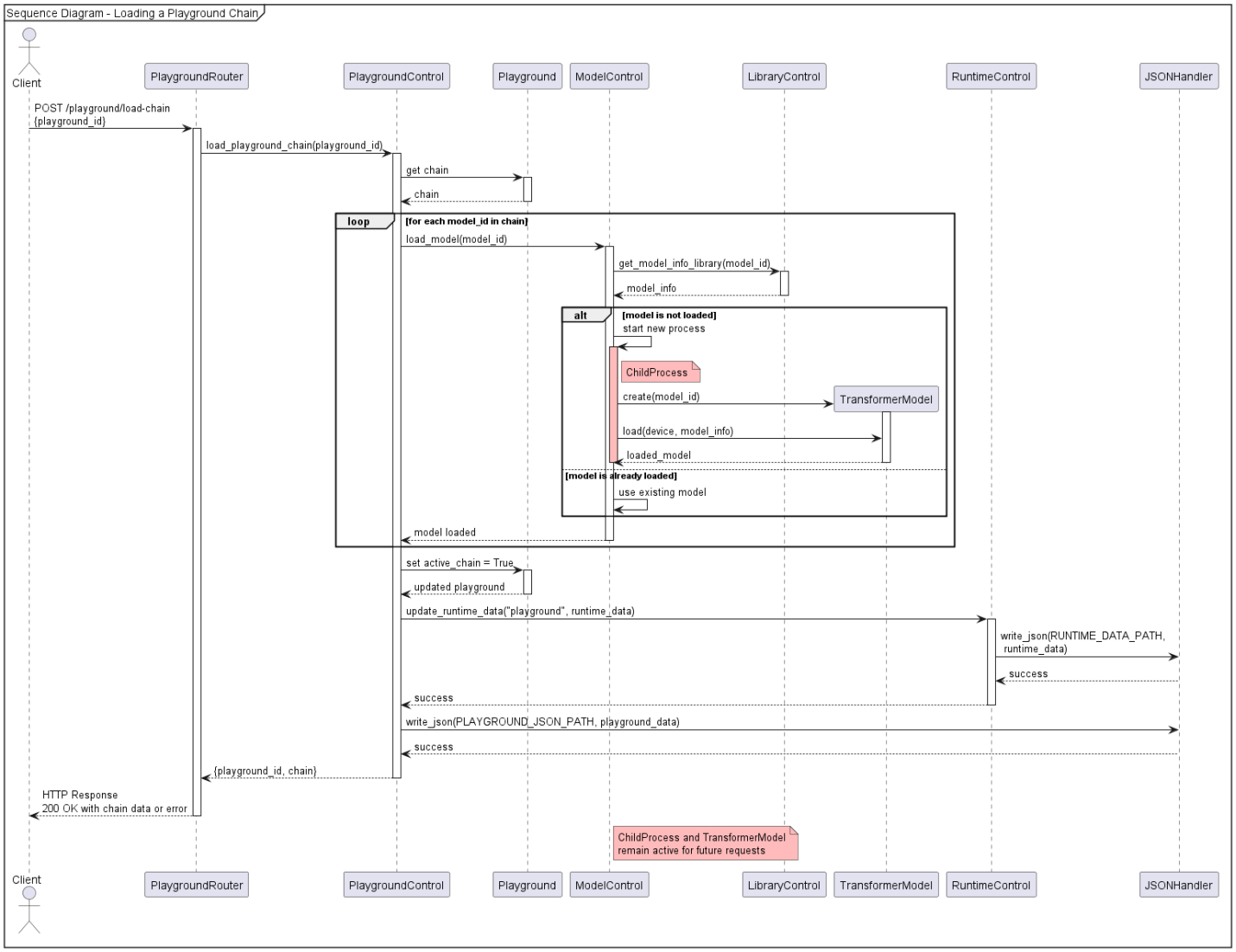


Figure 5.4: Loading Playground Chain

Sequence 2 (Figure 5.4) illustrates the interactions that occur when a client (via the UI or API call) requests to load a playground chain within the system. A playground chain represents a series of models loaded in a specific configuration.

This diagram was specifically chosen because it not only details the process of loading a playground chain, but also, by extension, provides valuable insight into how individual models are loaded within the system.

The sequence begins with the PlaygroundRouter, which receives a request to load the playground chain. This request is forwarded to the PlaygroundControl, which manages the playground's state. The PlaygroundControl retrieves the chain data from the Playground entity, iterating through each model in the chain.

For each model_id in the chain, the ModelControl class checks whether the model is already loaded. If the model is not loaded, ModelControl interacts with LibraryControl to retrieve the necessary model information from the library. The model loading process follows the same pattern as seen in Sequence 1, where a separate process is created to instantiate the TransformerModel. This is represented by the red activity bar, indicating that the TransformerModel instantiation occurs in a child process.

It is important to note that the TransformerModel lifeline starts at the point where the child process is invoked, which is why its lifeline is shorter compared to the other classes. Unlike in the previous diagram (Figure 5.3), the TransformerModel persists once it is loaded, as it is required for subsequent model inference tasks. The child process containing the TransformerModel will therefore remain active and will only be terminated by a separate client request.

Once all models in the chain are successfully loaded, the PlaygroundControl sets the chain as active and updates the playground data. This is followed by an update to the runtime data, managed by RuntimeControl. Both the updated playground and runtime data are written to their respective JSON files using the JSONHandler utility class.

Finally, the system sends a success response back to the client, confirming that the playground chain has been loaded with the appropriate models and configurations. The child process and TransformerModel remain active, ensuring that subsequent operations, such as model inference, can take place.

5.5.3 Sequence 3: Inferencing A TransformerModel

Sequence 3 visualises the interactions that take place when a client (via the UI or an API call) submits a request to perform inference using an already-loaded TransformerModel. This diagram is essential for understanding how the system processes inference tasks using models that have been previously loaded and are persistent in memory.

The sequence is initiated when the client sends an inference request to the system. This request is handled by the ModelRouter, which passes it to ModelControl. At this stage, ModelControl is responsible for coordinating the inference operation with the relevant model, in this case, a TransformerModel that is already loaded into system memory from a prior operation (such as the Load Playground Chain sequence).

ModelControl then sends the inference request to the child process where the TransformerModel instance resides. This child process is represented by the red activity bar in the diagram, indicating that inference is handled within the separate process. The TransformerModel processes the inference request and generates the result, which is then returned to ModelControl.

Once the inference result is received from the child process, ModelControl forwards it to the ModelRouter. The ModelRouter then sends a success response back to the client, along with the result of the inference operation.

This sequence highlights a critical design choice: the separation between the loading and infer-

ence processes. By comparing this to the loading steps seen in Sequence 2 (where models are loaded into memory), it becomes clear that once a model is loaded, there is no need to reload it for each inference request. This separation ensures that models remain available for reuse, which significantly improves system performance and efficiency. Additionally, this design enhances the overall user experience by reducing latency during inference operations, allowing for faster responses to client requests.

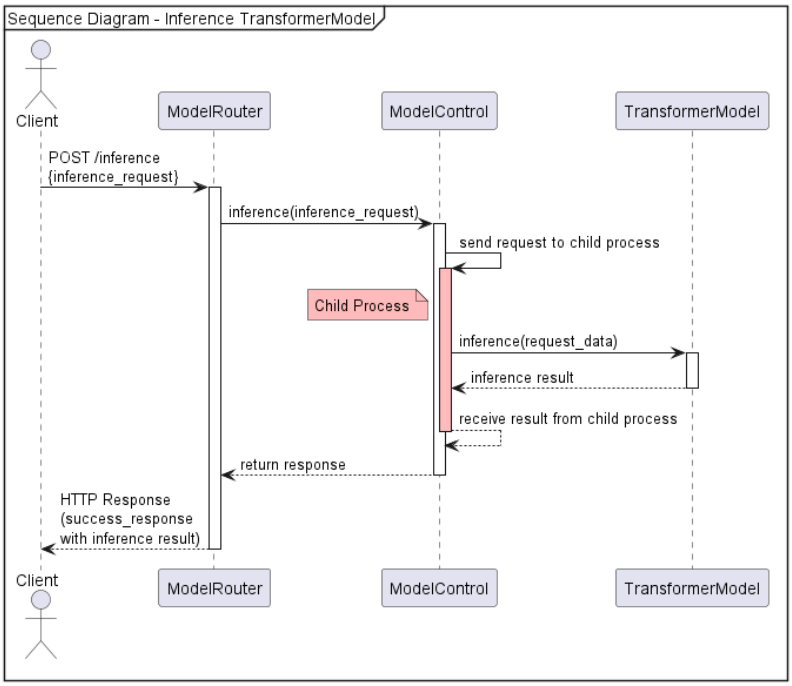


Figure 5.5: Inferencing TransformerModel

5.6 Frontend design

The decision to use .NET MAUI as a C# framework for UI development meant that the design pattern for the frontend was predefined, adhering to the Model-View-ViewModel (MVVM) architecture. This pattern separates the frontend into three distinct but interconnected components:

- **Model:** The Model represents the application's data and business logic. It is responsible for managing data, processing requests, and maintaining the current state. The Model operates independently of the UI, ensuring that no UI-related logic is handled at this level.
- **View:** The View is the user interface component that displays data to the user. It is responsible for presenting the information received from the ViewModel and accepting user input. The View itself does not handle business logic, instead relying on the ViewModel for data and state management.
- **Controller:** The ViewModel serves as the intermediary between the View and the Model. It processes user actions, manages the UI state, and retrieves or updates data from the Model. The ViewModel binds directly to the View, ensuring that the UI is updated reactively whenever the data changes.

Given this predefined MVVM framework, design choices for the frontend were somewhat constrained. However, the integration of the Backend with the Frontend remained a critical decision point.

To achieve this, a set of Service classes were implemented to mirror the Router classes from the backend. Each service class corresponds to a specific data type, following a consistent pattern. In particular, PlaygroundService, SettingService, DataService, ModelService, and LibraryService were created to handle different types of backend interactions and are represented in Figure 5.2. These service classes encapsulate reusable methods that interface with the backend API endpoints, allowing seamless integration of backend functionality into the frontend.

By adopting this structure, the service classes provide a modular and maintainable approach. The methods within these service classes can be directly accessed by the ViewModel to fetch or update the necessary data. This design ensures that all API requests are handled in a single, well-defined location, preventing API logic from being scattered across the frontend codebase.

Additionally, the service classes play a secondary role: transforming the JSON data returned from API requests into a format usable by the frontend. Since C# is a strictly typed language, the JSON responses need to be deserialised into Model (MVVM) objects, ensuring they can be processed and displayed by the application.

Figure 5.6 represents a high-level frontend class diagram abstraction, illustrating how the generalised ServiceClass interacts with the MVVM framework. This also expands on the details of the UI abstraction, previously represented by the stereotype `<<Abstraction>>` in Figure 5.2, with the UI encapsulated inside the Frame by the same name. The labels Data Binding and Data Access provide a high level interpretation of the interactions between the MVVM components, whilst the Uses relationship provides a more standard interpretation of class interaction. The ViewModel directly uses the ServiceClass to fetch backend data, which the ServiceClass then deserialises into a Model object which the ViewModel can use to update the View.

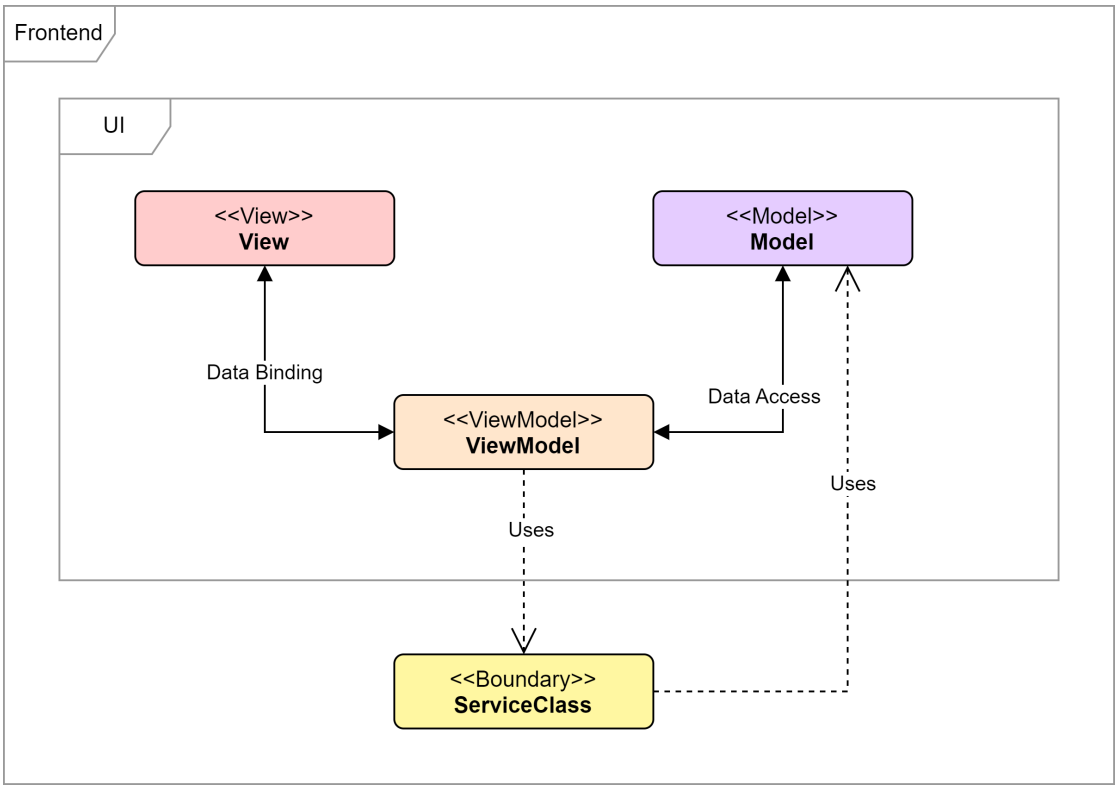


Figure 5.6: Frontend class Interaction Diagram

The choice to use a high-level abstraction in Figure 5.6 is intentional, as the key methods of the

ServiceClass are already defined within the backend Router classes shown in Figure C.24. Creating a fully detailed class diagram for the frontend would be impractical given the large number of ViewModels and Views. Moreover, since .NET MAUI is an established framework adhering to the MVVM design pattern, there are few design choices to make in this area. Most methods focus on UI rendering and interaction, with little need for custom architectural decisions. This reliance on a structured framework and predefined backend methods justifies the high-level view, providing clarity without unnecessary complexity.

5.7 Data Schema

As outlined in Research: 3.5.7 Data Structure, direct JSON interaction was selected as the preferred method for data storage. While a strictly defined structure is not necessary with this approach, a minimum set of attributes needed to be established to ensure the system functions as required.

This section details the four key JSON data files essential to the class structure defined earlier. These files are closely aligned with the entity classes, such as those representing playgrounds and models, and are based on the necessary data inferred from the class structure. The first step in defining these files was identifying the minimum data required to represent each entity.

5.7.1 model_index.json

For the model classes, the system requires that each model entry contains the minimum attributes shown in Listing C.1. The final two attributes, requirements and config, are specifically necessary for text-generation models like TransformerModel entries but are not always required for other models. With these attributes defined, a general index file called model_index.json was created. The key distinction between an individual model and its final representation in this index is that the model_id is moved outside the main content body and used as a primary key. This design choice simplifies data retrieval during implementation and allows the system to efficiently access model entries. And is visible in Listing C.2.

Each model_id serves as the key for the corresponding model's data, excluding the model_id itself within the content. This structure ensures that future model additions can integrate seamlessly with the application logic.

5.7.2 library.json

The second key data file is library.json (Figure C.3, which represents the models after they have been downloaded. This file extends the structure of the models found in model_index.json, adding crucial attributes necessary for managing the loading of downloaded models. While the structure is largely identical to that of model_index.json, each model entry in library.json includes four additional attributes.

These additional attributes are essential for the system to load and customise models after they have been downloaded. The auth_token enables access to certain gated models loading, while the base_model allows for model customisation, referencing the original model from which a customised version was created. The dir attribute points to the local directory where the model is cached, ensuring it can be easily loaded into memory. Finally, is_customised flags whether a model has been modified after its original download.

Although some models, such as WatsonModel, do not require downloading due to their online nature, the system is designed to accommodate both online and offline models. Even for online models, the system can extract and store relevant information from their associated services during the model's ”download” process, ensuring a consistent approach across all model types.

5.7.3 playground.json

The third key data file is playground.json, which contains a dictionary of playgrounds. Each entry within the file stores all the necessary information to represent a playground entity. Similar to the previous two files, the first level of playground.json follows a consistent structure. However, instead of using model_id as the primary key, each entry is indexed by a unique playground_id.

This consistency in structure ensures ease of development, allowing methods to be implemented in a similar fashion across different data sources. Nested within each playground_id are three main attributes that enable playground interactions: description, models, and chain, which are described in Listing C.4.

The ”models” sub-dictionary represents all the models that have been added to a given playground, containing a nested dictionary of model_ids. Each model entry includes specific information extracted from library.json. A deliberate design choice was made to include this small data excerpt within the playground rather than referencing it directly from library.json. Although this introduces some data redundancy, it simplifies the process of packaging the information for API requests to the frontend. For instance, the is_online attribute is not functionally relevant to the backend but is included to generate a UI element in the frontend. This approach reduces the number of file reads, potentially improving operational efficiency. The ”chain” attribute contains an ordered array of model_ids from the models attribute, defining the flow of the model chain within the playground.

5.7.4 runtime_data.json

The fourth and final data file in this section is runtime_data.json (Figure C.5), which was introduced to track model loading information, specifically which playgrounds are currently running a particular model. This is a critical consideration, as attempting to unload a model that is still active in a separate playground would result in an error. To prevent this, runtime_data.json was developed to keep track of these interactions.

Since runtime_data.json may evolve to accommodate future requirements, a data-specific key, playground, was included to denote that the runtime data currently pertains to playgrounds. Nested within this key are model_ids, representing models currently loaded into system memory within a playground chain. Each model_id contains an array of playground_id strings, representing all playgrounds that are currently utilising that particular model.

This structure ensures that a model can only be unloaded from system memory when there are no playground_ids associated with the respective model_id. By tracking these dependencies, the system can effectively prevent unloading errors, ensuring that models remain active as long as they are needed by any playground.

Chapter 6

Implementation

This chapter outlines the implementation of the requirements set out in Chapter 2, with a focus on how these were realised through the system design detailed in Chapter 5. Where appropriate, individual project requirements will be discussed in the context of the broader application to highlight their integration.

The chapter begins with an overview of the application's structure implementation, which serves as the foundation for effectively implementing the system design.

6.1 Application structure

To ensure a highly maintainable and organised codebase, a well-structured and logically laid-out file and folder hierarchy was essential. This structure was iteratively developed during the test application phase, briefly outlined in Chapter 3. The final implementation consists of a parent folder with three primary subfolders: backend, data, and frontend. This initial separation clearly defines the responsibilities of each part of the system, contributing to the maintainability of the application.

The frontend folder largely follows the default structure provided by .NET MAUI, with one key addition: a services folder containing the service classes, as illustrated in Figure 5.2. This folder encapsulates the service layer responsible for connecting the frontend to the backend.

The data folder houses the primary JSON data files, outlined in the Data Schema section. It also includes a downloads folder, which acts as a cache for downloaded models, with model-specific folders nested within.

The backend folder contains all the business logic of the application and can be viewed in Figure C.25 along side a detailed description of each of the folders. This folder is divided into several subdirectories: api, controllers, core, data_utils, models, playground, settings, test, and utils. It also includes a requirements.txt file, which lists all necessary libraries for the application to function properly.

As team leader, implementing a robust folder structure was of personal importance to ensure smooth integration of work across the team and to provide clarity in understanding the system design detailed in Chapter 5.

6.2 Application Entry Point: main.py

The main.py file serves as the central entry point to the application, with a reduced view shown in Listing C.6, which will be referenced throughout this section. Its primary responsibilities include launching the FastAPI server, instantiating the necessary control and router classes, and handling file creation.

A key insight during the development of class interactions was the need to ensure that only one instance of ModelControl persists across the entire system. This is critical because ModelControl is responsible for managing all currently loaded AI models, and having multiple instances could lead to inconsistencies, preventing interactions with models across different components of the application. For instance, PlaygroundControl relies on the same instance of ModelControl to function correctly, meaning it, too, must exist as a singular instance, loaded during application start. This implementation is demonstrated on lines 6 and 8 of the Listing.

Following the instantiation of the control classes and the necessary file creation (lines 10-12), main.py proceeds to instantiate the router classes, passing the relevant control class instances as parameters (lines 14-19). This is followed by setting up the API routes (lines 21-26) and entering the main program loop (line 28).

The implementation of custom Router classes deviates from the standard FastAPI convention (as outlined in FastAPI's documentation [11]) and represents a novel approach, which will be explored in more detail in the following section.

6.3 Router Classes

In FastAPI, the conventional approach for large applications involves defining separate files to group API routes, which are then included in the main FastAPI server using the app.include_router(route_file_name) method. Dependencies are typically injected using the dependencies=[dependent_classes/methods] parameter. However, this can become cumbersome when extracting the dependencies inside the route file for specific API requests, as they must be explicitly referenced each time. This is demonstrated in Listings C.7 and C.8, where an instantiated class must be referenced multiple times within the route function definitions.

An alternative approach was identified by examining the FastAPI APIRouter class definition/documentation and leveraging the add_api_route(*args) method to enable a more streamlined implementation. This approach involved creating a custom router class where APIRouter() is instantiated during the initialisation of the class. This allowed methods to be defined in a standard class format, and crucially, a single instance of ModelControl could be passed into the router class during instantiation. This instance is then referenced via an instance variable, such as self.model_control. A general example of this implementation is shown in Listings C.9, with its integration into main.py demonstrated in Listings C.10.

This implementation ensures that the methods of a single instance of ModelControl are available to all API endpoints within the router, while also providing a more consistent class structure across the backend of the application. This uniformity enhances maintainability and scalability.

Highlighting this design decision is essential, as the majority of the features described in the following sections depend on API requests constructed using this structure. Without this implementation detail, the API requests would not appear to adhere to any available structure documenta-

tion, potentially causing confusion when aligning the implementation with standard practices. Any feature implementation described in the following sections can be assumed to be triggered via an API request, and due to the extensive coverage in this section as well as detailed listings, specific API requests will not be explicitly detailed when describing feature access. Additionally, due to the separation of responsibility outlined in Chapter 5, the routes act purely as integration points for control class logic, meaning further detailed reference to them is unnecessary. Instead, a full listing of relevant API request endpoints for each Router class will be outlined in Appendix Section C.6.5.

6.4 Process Management

As outlined in the multiprocessing section, efficiently managing the loading and unloading of AI models in system memory was a critical requirement. By leveraging Python's multiprocessing library, a robust implementation was developed within ModelControl to handle this functionality.

When a request to download or load a model is received, ModelControl processes these via main thread methods such as download_model and load_model. These methods are responsible for retrieving the necessary data from the system, including the specific model class that will be instantiated within a child process. Using the multiprocessing.Pipe() function, a parent-child connection is established to allow communication between the processes. The relevant model data, child connection, and static methods defining the core functionality are then passed to the newly spawned processes. For download_model, this method is _download_process,
while for load_model, it is _load_process. Generalised pseudocode examples of the latter two can be found in Listing C.14 and Listing C.15 respectively.

Once invoked, these methods run independently of the main system thread, while still maintaining communication via the pipe. For instance, the _load_process function enters a loop that keeps the connection open, allowing it to handle tasks such as AI model inference requests. The loop remains active until a termination signal is received, at which point the connection is closed. Additionally, the use of multiprocessing.Lock() ensures controlled access to the loaded model, preventing conflicts during multiple request sequences.

While the pipe enables communication from child processes back to the parent process, an additional mechanism was required to keep track of the models loaded in their respective child processes. This was achieved using an instance variable nested dictionary, where each model's model_id serves as a key to store critical details such as the process object, connection, model class, and process ID, as shown in Listing C.16. This design allows the system to efficiently manage and communicate with the loaded models during runtime.

6.5 Hardware monitoring

Expanding on the concepts discussed in the previous section, the inclusion of the Process Identifier (PID) in the self.models dictionary allowed for the introduction of thread-specific hardware monitoring. By utilising the psutil library, it became possible to extract real-time CPU and memory usage data for each process. Additionally, the GPUtil library was employed to collect live GPU data on systems with CUDA support, enabling more comprehensive hardware usage monitoring. A simplified pseudocode implementation of this functionality is provided in Listing C.17.

6.6 TransformerModel and Data Customisation

The implementation of the TransformerModel class and its associated data requirements expands directly on the concepts discussed in Research Section 3.5.2 and System Design Section 5.7. During development, key data attributes were identified, leading to updates in the JSON structure for text-generation-specific models.

For instance, certain models require an authentication token for gated downloads, which was identified during the download and load implementation phases. This feature was subsequently incorporated into the TransformerModel class, reflecting text-generation-specific requirements and configurations as outlined in 5.7.1.

In addition to model-specific attributes, general configuration data necessary for downloading and loading models was identified. This included the classes mentioned in 3.5.2, such as transformers.AutoModelForCausalLM and transformers.AutoTokenizer, which enable access to text-generation models from the Hugging Face library. Given that the TransformerModel class needs to support a wide range of task types, a dynamic approach was taken to load the necessary libraries based on the model and task type. This is demonstrated in Listing C.18, which highlights dynamic model loading and the extraction of authentication tokens.

Text-generation-specific configuration options, such as system prompts and pipeline configurations, were also added to the model-specific configuration section within the model index. These settings are extracted during the model loading and inference processes to allow for customisable results. This implementation relied heavily on nested conditional statements to handle the extensive range of task types supported by the Transformers library, which posed a significant challenge. A detailed example of the resulting requirements and configuration attributes for text-generation models is shown in Listing C.19.

Through dynamic importing, the TransformerModel class is fully capable of implementing the BaseModel AbstractClass described in 5.3. Furthermore, by leveraging specific JSON data structures within the config and requirements keys, the class allows for a tailored implementation for text-generation models.

Due to the complexity and length of the full class implementation, it is recommended to refer to the supplementary materials for more in-depth context.

6.7 Optimisation and Quantisation

As discussed in the model selection section, hardware constraints imposed limitations on the size of models that could be deployed. Although smaller models were prioritised, they still present considerable demands on system resources. To mitigate these challenges, two key libraries, Accelerate and Bitsandbytes, were employed to optimise memory usage and enhance inference speed, thereby allowing the system to run models more efficiently on limited hardware.

Accelerate:
The Accelerate library provides a streamlined solution for optimising model deployment across available hardware, including both CPUs and GPUs. It reduces memory overhead by utilising mixed-precision execution, where calculations are performed in lower precision without significantly compromising accuracy. By employing the Accelerator class, device placement is automatically managed, ensuring that models are executed efficiently on the available hardware. This is especially

advantageous for this project, where minimal memory usage is critical, and no major modifications to the codebase were required. Additionally, the ability to distribute workloads across multiple devices ensures faster inference times, meeting the project's need to handle resource-intensive models on constrained hardware setups.

Bitsandbytes:
Bitsandbytes is another essential tool utilised in the project for optimising memory usage, specifically through model quantisation. By reducing the precision of model weights from standard 32-bit or 16-bit floating-point precision down to 8-bit or even 4-bit, Bitsandbytes significantly reduces the memory footprint of the models. This allows for the deployment of more capable models on hardware with restricted memory capacity, without a substantial loss in model accuracy or inference performance. For the AI Islands project, where efficient use of memory is paramount, Bitsandbytes enables the system to support larger models while maintaining acceptable performance levels, ensuring the backend can operate effectively within the given hardware limitations.

Listing C.20 demonstrates the general integration of these optimisation techniques, combining loading, downloading, inference, configuration, and quantisation. It directly expands on the research carried out in Section 3.5.2.

Chapter 7

Testing

The AI Islands project underwent extensive testing to ensure functional consistency and correct component interaction during development, as well as usability of the final application. These tests were conducted through Unit, Integration, and User Acceptance testing.

- **Unit testing** focuses on verifying that individual units of the programme, such as classes and methods, function correctly in isolation. An example of this would be testing the load method from the TransformersModel class.
- **Integration testing** ensures that different components of the system interact as expected. Rather than testing individual functionality, integration tests examine the flow of data between components to verify that they work together properly.
- **User Acceptance testing** (UAT) is the final phase of testing, where the application is evaluated by real-world users and stakeholders to ensure it meets the project's requirements. UAT is not technical but instead verifies that the system performs according to business requirements, paving the way for deployment.

In total, 243 Unit tests, 54 Integration tests, and 5 User Acceptance tests were performed, with a particular focus on testing text-generation AI models. The details of these tests will be discussed in the following sections.

In order to facilitate a consistent system state during each state as well ensuring the tests didn't interfere with general application ruining, test teardown was incorporated. This involved cleaning up after a test has been executed to ensure that no residual data, configurations, or states from the test interfere with subsequent tests or affect the application environment after testing is complete.

The solution involved creating a fixture within conftest.py, set to run during every test by using the autouse=True attribute. This fixture resets the data files to their pre-test state, ensuring there are no conflicts with remaining tests or subsequent application launches. The implementation is detailed in Listing C.26.

7.1 Unit Testing

As outlined in Figure C.24, the AI Islands backend system comprises numerous methods, many crucial to its core functionality. To ensure that these methods maintained their required function-

ality throughout development, Unit tests were created using the pytest library. The focus was primarily on testing core functions related to model and playground interaction, particularly where complex implementations were involved. Methods with minimal logic or those that follow a strict structure were generally not unit tested, as they primarily serve to route data and are more focused on component interaction. For example, the APIRoutes, which contain no business logic, simply facilitate access to the ControlClasses and were therefore not included in Unit testing.

As noted in the chapter introduction, a total of 243 Unit tests were created, covering methods from LibraryControl, ModelControl, PlaygroundControl, TransformerModel, and Playground. A folder containing the complete test suite is included in the supplementary materials, while the outcome of the Unit testing is provided in Figure 7.1. For the sake of brevity, this section will focus on a subset of these tests, specifically highlighting the downloading and inferencing of text-generation models using the TransformerModel class.

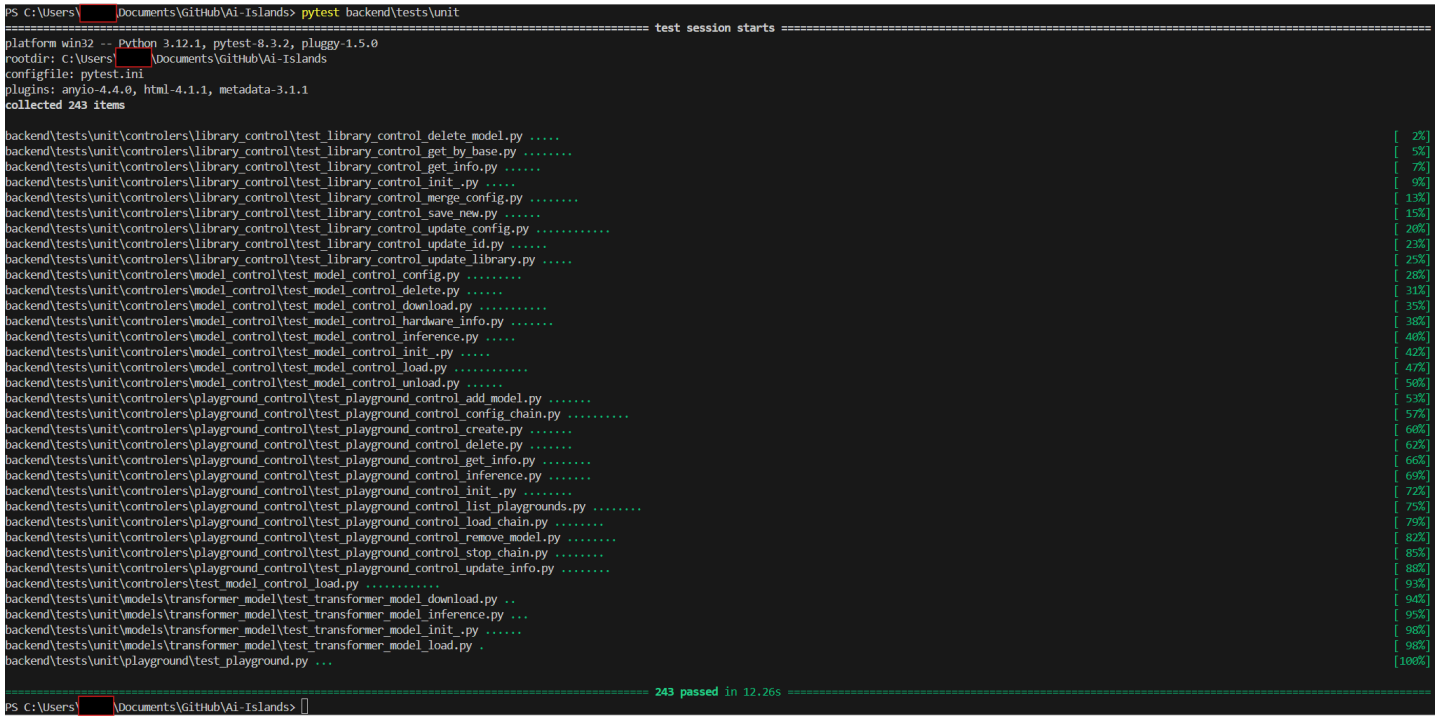


Figure 7.1: Unit Test Results

7.1.1 TransformerModel.download

Within the TransformerModel.download method, a call is made to the transformers library, which interacts with an external service. Since Unit tests are designed to isolate specific functionality within code, testing external service responses falls outside their scope. To handle this, Mocking and Patching were employed to simulate the expected return values of external services, a technique used extensively throughout the Unit tests in similar circumstances. Patching temporarily replaces a target object with a mock or specified object for the duration of the test, allowing external interactions to be bypassed with predefined return values. In the case of testing a successful text-generation model download, AutoModelForCausalLM.from_pretrained and AutoTokenizer.from_pretrained were identified as the two external service interaction points and were mocked using MagicMock from the unittest.mock library and patched during the test. This allowed the download method to execute without interruption.

Additionally, two other patches were used to simulate the creation of a download directory without

actually creating one—these involved patching os.path.exists and os.makedirs. Once the method execution was complete, assertions were employed to verify the success of the execution based on the mocked inputs and expected outputs. This included checking inputs from the mocked methods and the result of the download method execution, comparing them against expected values to ensure the method was functioning as intended. An excerpt of this Unit test can be found in Listing C.21.

Since methods may return different values based on varying inputs, each conditional statement leading to an output represents a different implementation of the function and must be tested accordingly. For example, in the download method, the presence of an authentication token represents a separate functionality and therefore required its own Unit test. More complex methods, such as TransformerModel.inference, contain numerous possible outcomes and consequently require a larger number of Unit tests to ensure thorough validation.

7.1.2 TransformerModel.inference

Testing the successful inference of a TransformerModel instance follows many of the same principles as the download example but with a different implementation. Instead of simply patching dependencies before execution, a fixture was used to create an instance of TransformerModel with its pipeline instance variable set to a MagicMock object. Fixtures are used to set up the required environment, resources, or data needed for the test, in this case returning a modified instance of TransformerModel.

Typically, the pipeline object would represent the fully loaded model in system memory, but actually loading and inferencing the model would be time-consuming and unnecessary for determining the success of the surrounding code. For this reason, it was mocked with a predefined output. Assertions were again used to test the expected input and output of the mocked pipeline. This implementation can be seen in Listing C.22 and for other TransformerModel.inference Unit tests please reference the supplementary material.

These two examples illustrate the core principles employed across the Unit tests, providing a reference point for understanding how the remaining tests function.

7.2 Integration Testing

Integration tests aim to verify that multiple system components are working together as expected, and are generally more complex to create. The AI Islands Integration tests were no exception, and their development will be outlined in this section.

As with the Unit tests, the Integration tests focused on the core functionality of the application. However, while Unit tests assessed the functionality of individual methods within classes (as presented in Figure C.24), Integration tests examined the interactions between components, as shown in Figure 5.1. The APIRoutes component was selected as the starting point for testing, as it integrates directly with the backend, simulating user and frontend requests to core application features such as model downloading, model loading, and playground chain inference.

Given the focus on testing component interactions involving Hugging Face text-generation models, direct interactions between the APIRoutes and the Utilities component were omitted. Instead, attention was placed on their interactions with the ControlClasses, EntityClasses, and their indirect interactions with Utilities.

The Integration tests were divided into two categories: Narrow and Broad integration tests (as defined by https://martinfowler.com/bliki/IntegrationTest.html). Narrow tests focus on a subset of component interactions, while Broad tests evaluate the functionality of multiple components in the context of a full feature request.

A good example of this distinction is found in the tests for model loading and model inference. These features were covered by Broad integration tests, which examined the flow from the ModelRoutes, through ModelControl, and down to the TransformerModel class, as illustrated in Figure 5.2. On the other hand, loading and inferencing a playground chain were considered Narrow tests, since both rely on the same method calls from ModelControl to TransformerModel. These tests only needed to validate the interactions between PlaygroundControl and ModelControl, with method interactions involving TransformerModel being mocked to avoid redundancy.

The combination of Narrow and Broad integration tests allowed for a faster test suite that maintained comprehensive coverage of all necessary features by reducing repetitive method executions.

Although Integration tests conventionally validate real system interactions (including interactions with external services), certain scenarios such as model downloading, loading, and inferencing were deemed impractical due to the resource-heavy nature and long wait times involved. Therefore, Mocking and Patching were employed to simulate external service calls within the TransformerModel download, load, and inference methods.

This posed a unique challenge, as access to these methods is handled via the multiprocessing library pipe defined in ModelControl, the necessity of which is explained in Section 3.5.4. When tests are executed, patches are applied to the current execution thread, but these patches are not passed to spawned child processes, effectively isolating TransformerModel from integration test patches applied in the parent process. Please reference 5.5.3 Sequence 3: Inferencing A TransformerModel, for explanation of system calls with child processes.

```
PS C:\Users\[redacted]\Documents\GitHub\Ai-Islands> pytest backend\tests\integration
============================================== test session starts ==============================================
platform win32 -- Python 3.12.1, pytest-8.3.2, pluggy-1.5.0
rootdir: C:\Users\[redacted]\Documents\GitHub\Ai-Islands
configfile: pytest.ini
plugins: anyio-4.4.0, html-4.1.1, metadata-3.1.1
collected 54 items

backend\tests\integration\library\test_library_full_index.py .                                          [  1%]
backend\tests\integration\library\test_library_full_library.py .                                        [  3%]
backend\tests\integration\library\test_library_get_model_info_index.py ...                              [  9%]
backend\tests\integration\library\test_library_get_model_info_library.py ...                            [ 14%]
backend\tests\integration\model\test_model_config.py ....                                               [ 22%]
backend\tests\integration\model\test_model_download.py ....                                             [ 29%]
backend\tests\integration\model\test_model_inference.py ....                                            [ 37%]
backend\tests\integration\model\test_model_load.py ....                                                 [ 44%]
backend\tests\integration\playground\test_playground_CRUD.py ..........                                 [ 62%]
backend\tests\integration\playground\test_playground_chain_inference.py .....                           [ 72%]
backend\tests\integration\playground\test_playground_chain_on_off.py ........                           [ 87%]
backend\tests\integration\playground\test_playground_config_chain.py .......                            [100%]

============================================== 54 passed in 73.47s (0:01:13) ==============================================
sys:1: RuntimeWarning: coroutine 'SettingsService.get_hardware_preference' was never awaited
RuntimeWarning: Enable tracemalloc to get the object allocation traceback
sys:1: RuntimeWarning: coroutine 'AsyncMockMixin._execute_mock_call' was never awaited
RuntimeWarning: Enable tracemalloc to get the object allocation traceback
PS C:\Users\[redacted]Documents\GitHub\Ai-Islands>
```

Figure 7.2: Integration Test Results

The solution involved creating subclasses with mocked methods and instance variables to simulate outputs where necessary, such as MockTransformerModel(TransformerModel) shown in Listing C.23. This allowed the use of patching to introduce the mock subclass before the multiprocessing call, ensuring the modified methods were available within the child process, as demonstrated by the fixture in Listing C.24. A final test under this structure can be seen in Listing C.25.

These solutions were applied throughout the integration tests, though due to their complexity, full listings are impractical to include in this report. For more detail, the complete test suite can be found in the supplementary material, while the outcome of the integration testing can be seen in Figure 7.2.

7.3 User Acceptance Testing

User Acceptance Testing was divided into two sections: Test-Group UAT and Stakeholder UAT. The former focused on representative end-users, while the latter demonstrated the final application to the project stakeholders.

7.3.1 Test-Group UAT

Test-Group UAT involved creating a well-defined task list designed to directly test the core features required by the project brief. Willing participants, representing possible target users, were identified and asked to use the AI Islands application to complete the task list. Each task was marked as either a pass or fail based on whether the participant successfully completed it. Any failures were accompanied by recorded comments for further analysis. Prior to testing, participants were provided with an explanation of the application's purpose and features. However, no visual demonstration was given in order to evaluate whether the application was intuitive to use, as specified by Non-Functional Requirement 'NFR-1'.

Suitable participants were select and their details can be read in Table C.5.

These participants were tested on the tasks presented in Table C.6, which comprehensively cover all the requirements associated with the project brief. The participants were asked to perform the tasks in the order listed.

Upon completing the task list, participants were invited to provide additional feedback through a discussion, aiming to gather qualitative insights not captured by the task results alone. These results are displayed in C.7.

The test tasks were completed with an overall success rate of 56/60 (93%) and a mean score of 18.7/20, indicating a high level of user acceptance.

Both P1 and P2 encountered no issues during the tasks, which aligns with their respective technical backgrounds. However, P3 struggled with tasks UAT-6 and UAT-16. While they successfully navigated to the correct pages, they lacked the technical knowledge to identify what an API request was. Additionally, P3 encountered difficulties with UAT-9, where they were unable to select the supplementary dataset from the RAG settings and their unfamiliarity with the technical terminology made them hesitant to continue, leading to a failed task. P3's feedback reflects this lack of confidence in certain areas and can be read in Table C.7.

The UAT process yielded valuable insights into how well the AI Islands application met user ex-

pectations across different skill levels. While the overall success rate of 93% underscores the effectiveness of the design, the feedback highlighted areas where further improvements can be made, particularly in simplifying technical language and improving UI accessibility for less experienced users. This feedback will be critical in refining the application during future version development, ensuring it delivers a seamless user experience for both technical and non-technical audiences.

7.3.2 Stakeholder UAT

Stakeholder UAT was conducted through a demonstrational presentation delivered to both industry and internal stakeholders on September 6th, 2024, as well as a live interactive demonstration presented to internal stakeholder Prof. Dean Mohamedally on September 3rd, 2024.

As the stakeholders were responsible for providing the project brief and refining it through discussions with the project team, they had an intimate understanding of the project's requirements and their own metrics for determining whether the final product met its objectives. For this reason, Stakeholder UAT did not require a task list to assess the requirements, as was used in Test-Group UAT. Instead, the presentation and demonstration served as opportunities to gather their feedback and opinions.

During the live interactive demonstration with Prof. Dean Mohamedally, he expressed a highly positive opinion of the application, stating, "I'm very, very happy and impressed with what you've got," and, following the demonstration of the playground features, added, "This is so phenomenal, you don't know how many charities and people that we're working with need this." His feedback on the UI was also very encouraging, particularly praising the use of .NET MAUI. Additionally, he was impressed by the future expandability demonstrated through the backend code implementation and structure. Overall, his comments indicated that he felt the project met all the key requirements and fulfilled the project brief.

The demonstrational presentation was created using Canva and can be viewed via the provided link [12]. It featured an introductory talk followed by a video walkthrough of a use case scenario. Feedback from the presentation was overwhelmingly positive. Key IBM industry stakeholder Prof. John McNamara expressed that he was thoroughly impressed with the results and offered personal praise. IBM's Global Programme Lead, Education and Workforce Development, Sonia Malik, also attended the presentation and provided very positive feedback, spending an additional 20 minutes afterwards discussing potential use cases.

Overall, the Stakeholder UAT provided the final assessment of whether the project met the established requirements. Based on the feedback received, it is clear that the project was viewed as a strong success by all stakeholders.

Chapter 8

Conclusion

This chapter will evaluate the projects success against the original project goals and personal aims outlined in Chapter 1, Section 1.2. It will then proceed to carry out a critical evaluation of the work produced before concluding with ideas for future work and development.

8.1 Achievements

8.1.1 Project Goals Evaluation

While the Unit and Integration testing discussed in Chapter 7 provided a structured approach to verifying the functionality of the application, they did not assess whether the functionality itself aligned with the project's overall goals. In contrast, the User Acceptance Testing (UAT) not only offered insights into the combined system's functionality but also served as a direct means of evaluating the project against the requirements outlined in Chapter 2, Section 2.6 and consequently the overall project goals.

Therefore, alongside a personal analysis of the final project, the UAT results will be used and referenced throughout this evaluation.

- **Lowering the Technical Barrier for AI Adoption** The Test-Group UAT demonstrated that the application was accessible to users across varying levels of technical expertise. The UAT tasks involved downloading, loading, and inferencing with an offline text-generation model, tasks that would be practically impossible for users with little to no technical experience, such as participant P3. Although P3 encountered difficulties with some of the more technical tasks, they were still able to meaningfully interact with an offline text-generation model. This indicates that the application successfully lowered the barrier for entry, fulfilling this project goal.
- **Improving the Affordability of AI Solutions** While no specific tests were conducted to assess the affordability of the AI Islands solution, this goal was inherently achieved through the successful development of the application. As outlined in the introduction, online services can often incur significant costs, and traditional offline solutions tend to require expensive hardware. The AI Islands system addresses both challenges by removing the reliance on online services and offering a curated list of smaller text-generation models (see Section 3.3), while also implementing quantisation options to further reduce hardware requirements.
- **Offering Privacy-Conscious AI Tools** As with affordability, privacy concerns were addressed through the successful implementation of specific features. The inclusion of downloadable

offline text-generation models in the AI Index enables users to store their data and queries locally, ensuring greater control over their privacy. Additionally, the implementation of Retrieval-Augmented Generation (RAG), as evaluated in UAT-09 and UAT-10 (Table C.6), allows users to query their own data without relying on external services. These features clearly demonstrate that this goal was successfully met.

- **Supporting Scalability and Integration** Although UAT did not specifically assess the system's integration with external applications and systems, the project team conducted thorough end-to-end testing via the developed API access feature using PostMan. This simulated real-world external application integration via API requests. Furthermore, the successful use of the UI during UAT indirectly demonstrated this feature, as the Frontend system is connected to the Backend through the same API requests that would provide access for external applications. In this sense, the Frontend effectively acts as an external application to the Backend, proving the goal was met.
- **Customisation and Flexibility** This goal, being feature-specific, was sufficiently demonstrated through the successful testing of the Playground chain (UAT-13, UAT-14, UAT-15, UAT-17, and UAT-18), as well as model configuration features (UAT-07, UAT-08, and UAT-11) for text-generation models. These tests highlight the system's ability to customise and configure models according to user needs, validating this goal.

8.1.2 Personal Aims Evaluation

The evaluation of the personal aims outlined in Chapter 1, Subsection 1.2.2, cannot be assessed in the same way as many of the project goals. Instead, their evaluation is based on a combination of reflective analysis and evidence demonstrated through project contributions.

- **Expand Knowledge of AI and Machine Learning** The research into text-generation AI models, thoroughly detailed in Chapter 3, significantly deepened the understanding of the AI landscape. This research introduced new concepts such as tokenisation and the impact of model training choices on parameter sizes. Additionally, the hands-on experience gained during the implementation phase further expanded knowledge of AI models. Coordinating with team members to integrate the system design also required a broader understanding of the models chosen to meet their individual briefs. Combined, this project has clearly contributed to expanding knowledge in this field.
- **Enhance Software Engineering Skills** As demonstrated in Chapter 5, significant effort was invested in planning and designing the AI Islands system backend. Building on the experience gained during the COMP0071: Software Engineering module, this project provided a valuable opportunity to apply those skills in a practical setting. By employing the Entity-Controller-Boundary (ECB) architectural pattern in Python, Object-Oriented design principles were fully leveraged to create a modular, maintainable codebase. The freedom to develop the project from the ground up allowed for refinement of software engineering skills. These experiences support the successful achievement of this personal aim.
- **Develop REST API Expertise** Prior to this project, experience with REST APIs was limited, having been briefly encountered during the COMP0067: App Engineering module. The AI Islands project greatly expanded understanding through the extensive development of APIs to integrate the backend system. In particular, the novel implementation of FastAPI routes, outlined in Chapter 6, demonstrates a deeper understanding of API systems. Furthermore,

the design and implementation of the frontend ServiceClasses provided valuable insight into how API requests are processed and utilised. Taken together, these experiences strongly suggest that this personal aim was successfully achieved.

- **Improve Understanding of Agile Frameworks and GitHub Kanban** As outlined in Chapter 1, Section 1.3, research and analysis were conducted to identify a suitable project management framework. This ultimately led to the adoption of a custom implementation of Agile methodologies that suited the team's needs. This approach improved development efficiency through task prioritisation and enhanced team collaboration. Overall, this demonstrates that the personal aim of improving project management understanding was sufficiently met.
- **Build Leadership and stakeholder Management Skills** Team leadership provided valuable experience, particularly during the initial stages of the project, when leading stakeholder discussions and requirement realisation. This process, detailed in Chapter 2, ensured a clear and consistent understanding of project requirements across the team. As the project transitioned into the development phase, a more hands-off management approach was adopted, guiding overall system development and offering support as needed, while ensuring all team members remained on track. This experience successfully contributed to the development of leadership and stakeholder management skills.

Conclusion of Personal Aims Evaluation Overall, the AI Islands project provided an excellent opportunity to progress with personal aims, with the individual evaluations indicating that all of the identified aims were successfully achieved.

8.2 Critical Evaluation

While the project has been widely considered a success, as indicated by the Stakeholder UAT feedback, there are several limitations that should be addressed in future development.

Streaming Response: Despite the sophisticated integration of APIs within the application, one notable feature remains absent when interacting with large language models (LLMs) for text generation. Reply times can often be slow, and LLMs support the ability to stream responses in chunks, improving response times and enhancing the user experience without affecting the overall turnaround time. Unlike standard FastAPI requests, incorporating FastAPI's StreamingResponse would allow this feature to be implemented, alleviating this limitation.

Resource Constraint Handling: The application allows users to load multiple models into system memory. On hardware-constrained devices, this can cause issues if too many models are loaded simultaneously, potentially leading to crashes or interruptions during inference. Currently, there is no mechanism in place to prevent users from overloading their devices, leaving it to their discretion. Future versions should implement safeguards to manage system resources more effectively.

Chain Constraints: The Playground chain feature allows users to create customisable AI workflows. However, it is currently limited in that middle-layer models are constrained to text-to-text interactions. This limitation restricts the flexibility of model combinations and should be addressed in future updates to expand the range of model chaining possibilities.

System Progress Information: Given that many of the models available in the system are large, downloading and loading them can take a significant amount of time. The system cur-

rently provides feedback during the download process by displaying terminal output. However, no such feedback exists for the loading phase. Implementing the aforementioned StreamingResponse module from FastAPI could provide real-time feedback during model loading, offering a more responsive and user-friendly experience.

8.3 Future Work

This section explores potential ideas for future development to further enhance the project, rather than focusing solely on limitations.

Model Containerisation: As outlined in Section 3.5.4, the current iteration uses child processes to manage individual model loading. While this is a viable solution, integrating containerisation through Kubernetes would allow the system to better adapt to different devices and enhance compatibility for various use cases. Containerisation would also enable more advanced hardware and performance monitoring, with each container capable of providing individual performance data. This would facilitate future development into application-specific benchmarking.

MongoDB Integration: As the application expands to include a larger number of models in the AI Index and additional features, it would be beneficial to integrate a database management system such as MongoDB. This would maintain the flexibility of the existing JSON file structure while offering more sophisticated data querying capabilities.

Server Deployment Compatibility: Currently, the AI Islands system functions as a desktop application, with external application integration limited to the local machine. However, the extensive API integration allows the backend system to operate independently. By incorporating containerisation through Kubernetes and implementing MongoDB for data management, the system could be deployed on either a local or external-facing server.

A modified version of the frontend interface could serve as a local client for the server, enabling multi-user functionality. This would be particularly advantageous for small businesses or educational institutions, where a single, more powerful system could be shared among multiple users. The inclusion of user and administrator accounts would allow for flexible control over system access. Administrators would be able to manage feature availability, such as deciding which users have access to model selection, testing, loading, or configuration settings. This feature access control would also extend to API integration, enabling administrators to grant or restrict API access based on user roles or project requirements.

The development of user accounts would not only support role-based access control but also provide the ability to monitor and manage system resources more effectively, ensuring that the system remains efficient for all users. This approach would create a more centralised, scalable solution while maintaining compatibility with the original project brief, enhancing the system's flexibility and broadening its potential for larger-scale deployments.

AI Index Updates in the UI: Introducing a UI feature that enables users to add newly available models from supported sources would significantly improve the maintainability of the application. This addition would reduce the technical expertise currently required to manually update the AI Index through direct JSON file editing, making the process more accessible and user-friendly.

References

[1] David Patterson, Joseph Gonzalez, Quoc Le, Percy Liang, Dan Munguia, Daniel Rothchild, Richard Socher, Jeffrey Dean, and John Hennessy. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021. Available at: https://arxiv.org/abs/2104.10350.

[2] The hugging face model index, 2024. Available at: https://huggingface.co/models.

[3] Gradio website, 2024. Available at: https://www.gradio.app/.

[4] Openai playground, 2024. Available at: https://platform.openai.com/playground/chat.

[5] Kubeflow website, 2024. Available at: https://www.kubeflow.org/.

[6] Github, 2024. Available at: https://github.com/.

[7] Tensorflow hub, 2024. Available at: https://www.tensorflow.org/hub.

[8] Pytorch hub, 2024. Available at: https://pytorch.org/hub/.

[9] Hugging face llama recipes, 2024. Available at: https://github.com/huggingface/huggingface-llama-recipes.

[10] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. Available at: https://doi.org/10.48550/arXiv.2001.08361.

[11] Fastapi documentation: Bigger applications, 2024. Available at: https://fastapi.tiangolo.com/tutorial/bigger-applications/.

[12] Canva stakeholder presentation, 2024. Available at: https://www.canva.com/design/DAGPzSvk6s4/ArmpHDlblp7czf21vFEH-g/edit.

Appendix A

Deployment Manual

This manual outlines the steps required to access and set up the AI Islands System. As the current version has not been compiled, the necessary setup instructions are detailed below.

Prerequisites Visual Studio Community 2022: Ensure Visual Studio is installed with the .NET MAUI workload selected. A tutorial for this setup can be found at the following link: Install .NET MAUI.

CUDA: For offline text-generation models, an NVIDIA GPU supported by the CUDA toolkit (version 12.5) is required. You can download it here: CUDA 12.5 Download Archive.

Python: The latest stable version of Python must be installed. It can be downloaded from: Python Downloads

GitHub Repository: To run the application, clone or download the AI Islands project from GitHub: AI Islands Repository

The branch this report is detailing includes the individually developed test files and is called alt_tests. This is also the version provided in the supplementary material.

Please note that while the application has been tested on Windows machines, it may be possible to run it on Linux systems if the required dependencies are met. However, this manual covers the setup process for Windows only.

Setup Instructions

1. Download and extract (or clone) the GitHub repository to your preferred location.
2. Navigate to the README file included in the repository.
3. Follow the setup instructions provided in the README to complete the installation.

As the project may undergo future development, the decision to represent the deployment guide through the repository README.md file, enables up to date setup information to be provided, regardless of when this report is read.

If repository access is unavailable, a copy of the README file can be found in the supplementary material codebase.

Appendix B

User Manual

This manual will cover the final AI Islands implementation, detailing how users can interact with the system to create and use an Offline Chat bot with RAG capabilities.

Please follow the instructions in the Deployment Manual to setup and launch the application first.

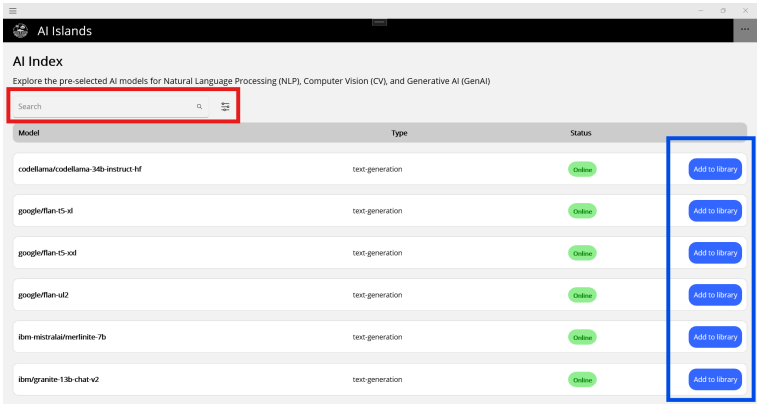


Figure B.1: AI Index

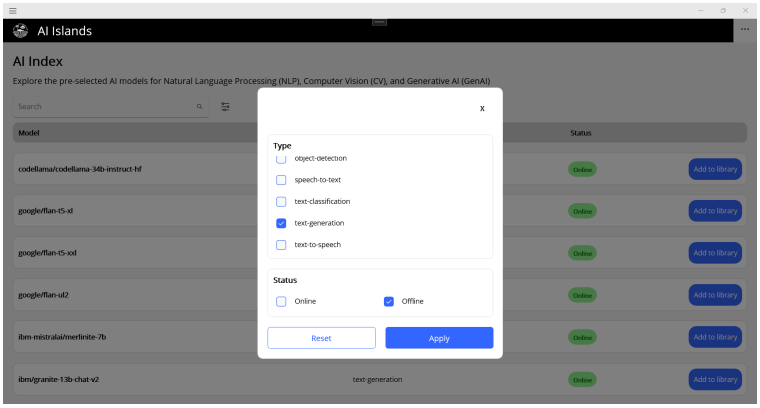


Figure B.2: Index Filtering

Figure B.1 shows the AI Index, highlighting the included search, filter and add to library buttons. Figure B.6 shows how the filtering can be used to select Offline text-generation models.

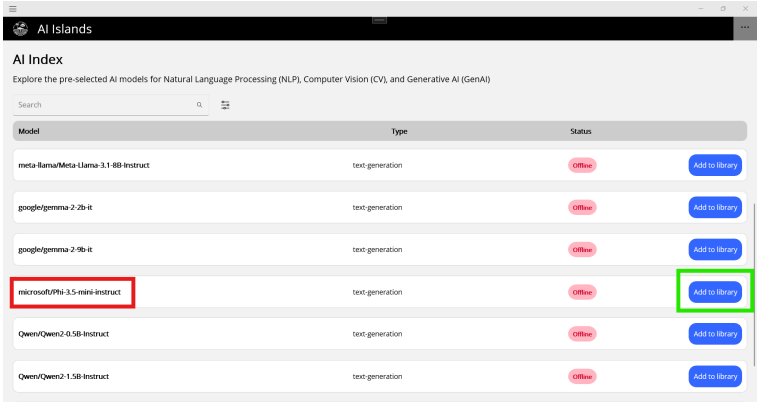


Figure B.3: Offline Text-Gen Model List

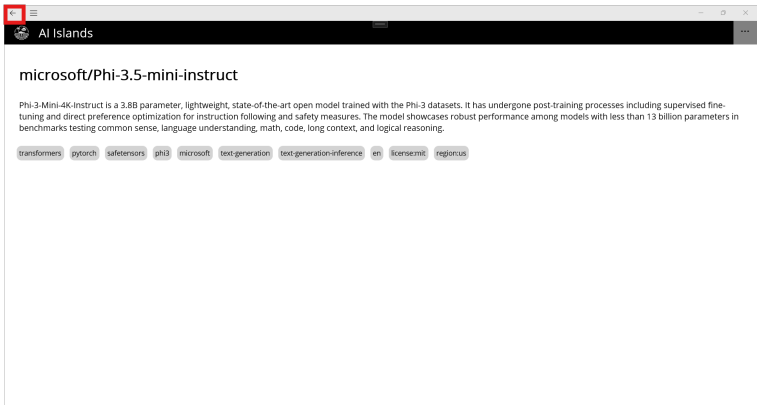


Figure B.4: Model Info

Figure B.3 shows the curated list of GenAI Offline text-generation models and highlights the access to model info in red and downloading/add to library in green. Figure B.4 shows the model

info page, with the return navigation button in red.

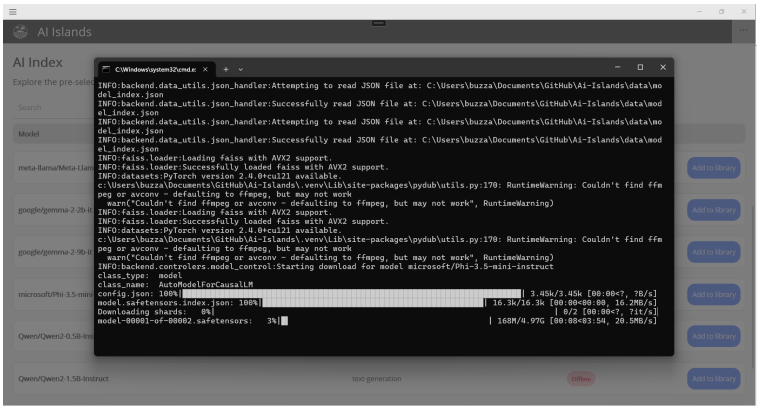


Figure B.5: Download Model View

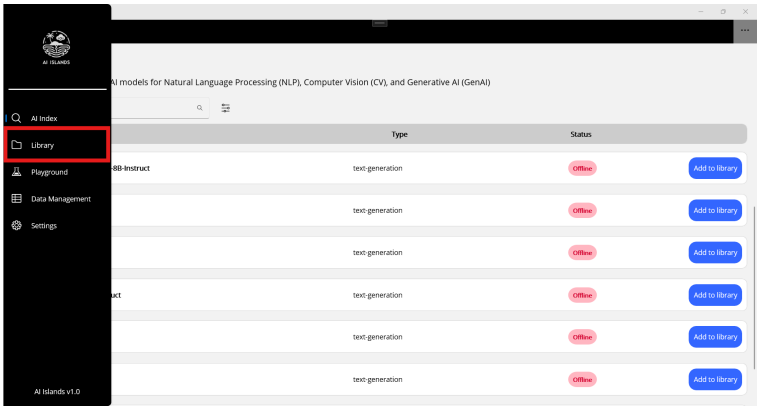


Figure B.6: Navigation Menu

Figure B.5 shows the view when a model is downloading, with terminal feedback to display download progress. Figure B.6 shows the navigation panel that can be used to access the various application pages, including the Library once a model has finished downloading.

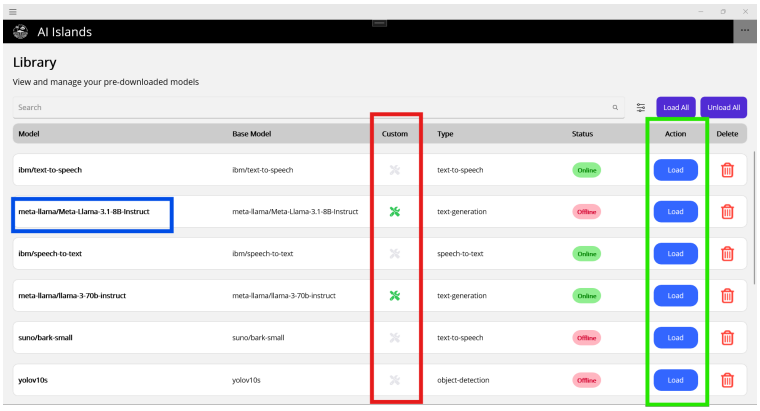


Figure B.7: Library Page

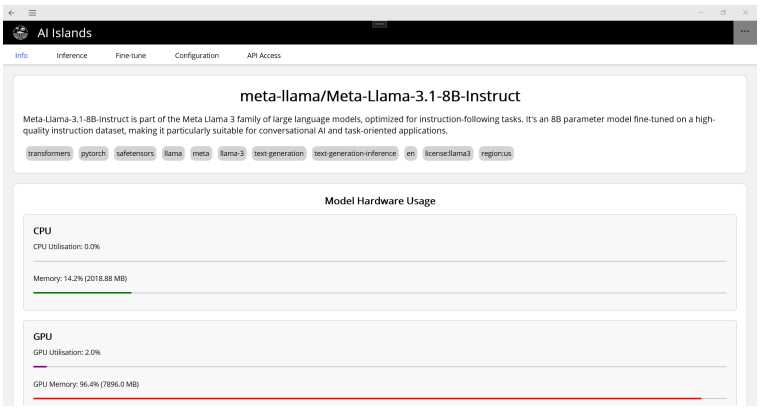


Figure B.8: Model Hardware Usage Information

Figure B.7 shows the library page, with custom model configuration indication highlighted in red. Highlighted in green are the load buttons which allow the user to load models into system memory, ready for inference. By selecting a model, highlighted in blue, the user will be taken to the model page where they can see model information such as hardware usage when loaded as shown in Figure B.8. From this page they are able to navigate to the other model interaction tabs as listed in the top bar.

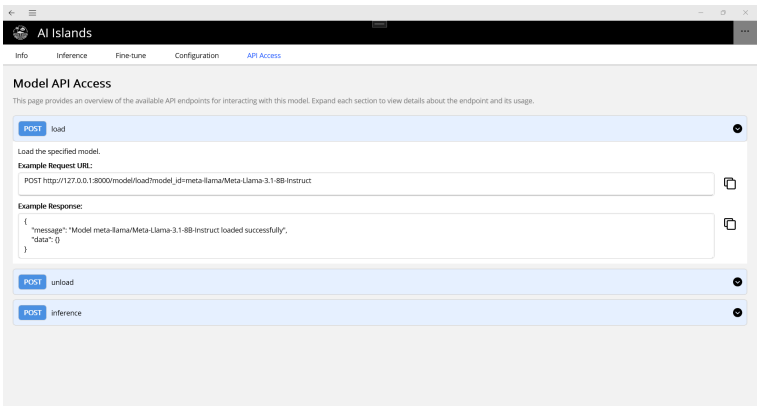


Figure B.9: Model API Access

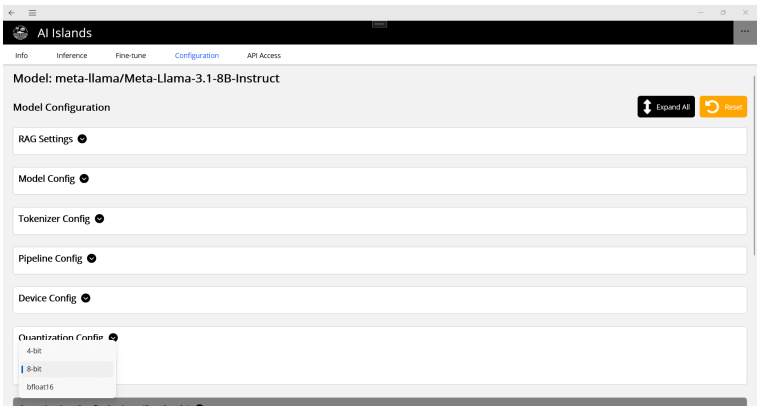


Figure B.10: Text-Generation Model Configuration

Figure B.9 demonstrates the API Access page for an individual model, featuring options for loading, unloading and inferencing, while Figure B.10 displays the Configuration page where users are able to select from many different available configurations, in this case showing quantisation options.

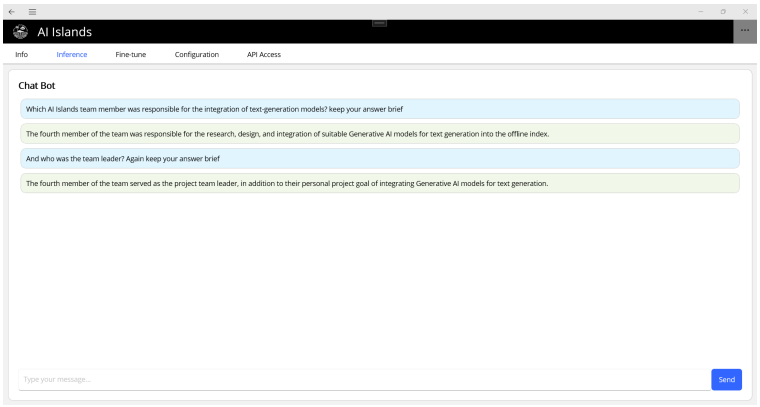


Figure B.11: Chat Bot Inference

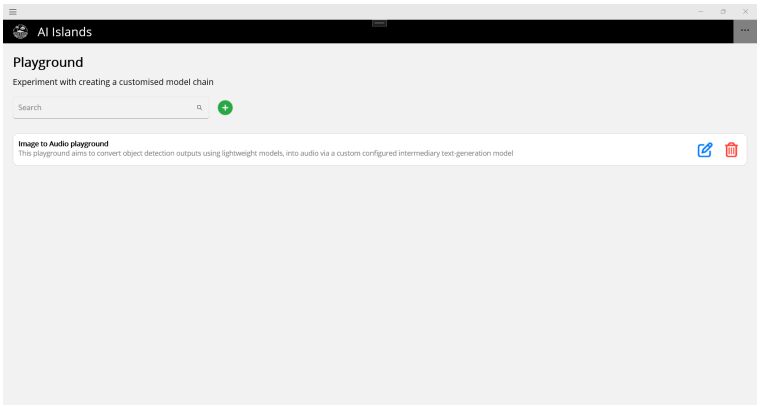


Figure B.12: Playground Page

Figure B.11 displays an example of inferencing a chat bot configured text-generation model (This option is available in the Configuration page). It also shows the included RAG features, with the model successfully able to answer questions based on the provided Motivation section of this report.

Using the previously shown navigation panel, the user is able to access the playground page shown in Figure B.12, in this case featuring an example playground. Here the users are able to search for, add and delete playgrounds, as well as update their names and descriptions. These features are accessible by the intuitive buttons visible.

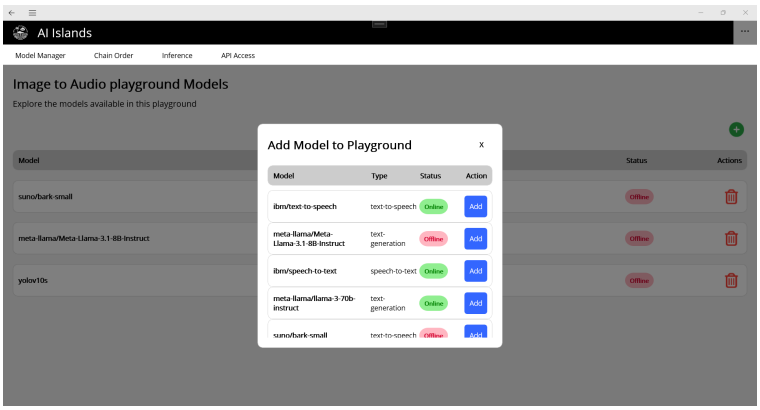


Figure B.13: Add Models to Project

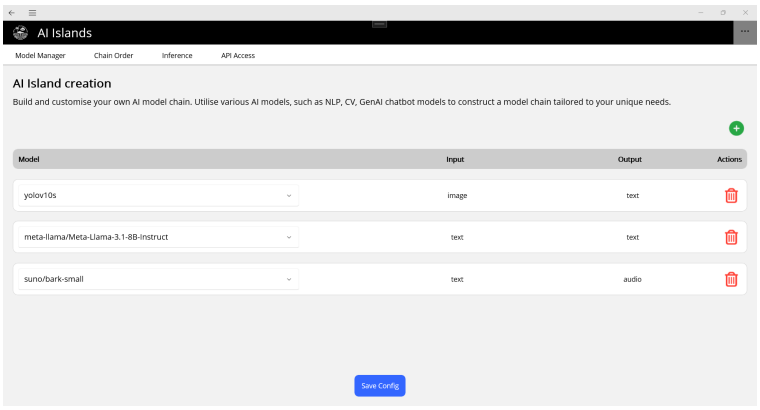


Figure B.14: Chain Configuration

Figure B.13 shows the Model management page of the playground, which the user will be presented with when clicking on a playground from the page shown in Figure B.12. In this case the Add Model overlay is open, which is accessed via the add button on the right hand side of the page. This shows how users can add downloaded models from their library directly to a Playground, ready to be implemented in a model chain.

Figure B.14 shows the Chain Order page, which allows users to create a model chain. Users can add models that they have previously added in their Model Manager page by pressing the green add button on the right hand side. This will create a new row in the list with an unpopulated drop down box on the left hand side under the Model column, from which users can select their models. This allows users to quickly swap out models in the chain. Once a chain has been defined, pressing the Save Config button will verify, and if valid save the chain.

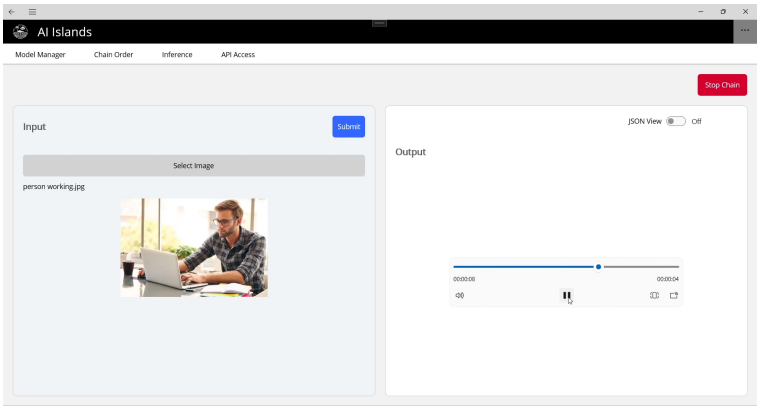


Figure B.15: Playground Inference

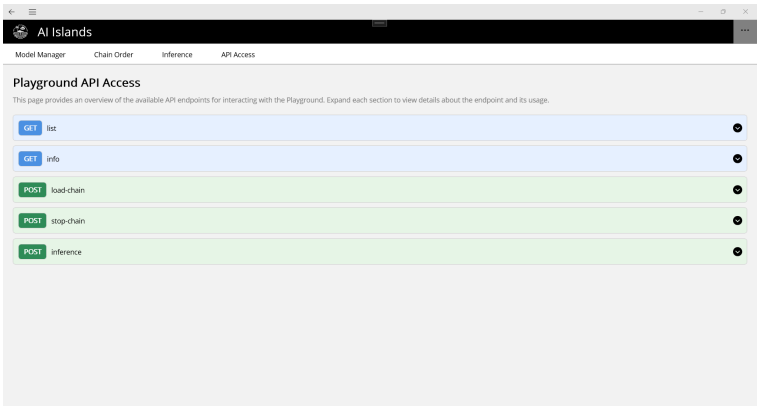


Figure B.16: Playground API Access

Figure B.15 shows the Playground Chain Inference page, where users can load and unload their previously saved chain configuration using the button on the top right hand side of the page. The submit button on the Input frame allows users to send their inference requests for processing by the chain, in this case displaying the input and output view of the chain configured in Figure B.14.

Figure B.16 shows the API Access page for the Playground features. Functioning in much the same way as the Model API Access page, it allows users to implement their custom chain configurations using the provided APIs.

Appendix C

Resources

C.1 Requirements Resources

C.1.1 Persona Resources

Computer Science University Student

"I'm excited to explore AI, but I need hands-on experience to really understand how these models work in practice."

Mark is a first year BSc Computer Science student looking to incorporate AI features into a project he is working on. He hasn't had any formal teaching with AI models yet, but is enthusiastic about the topic and understands the basic concepts through personal research. He is unable to afford AI paid services and is therefore searching for a free service that allows him to quickly download, test and implement the appropriate AI model into his project. He does have access to a reasonably fast laptop with a mid range GPU.

Name Mark Corrigan
Type User
Role University Student

Motivations

- Gain hands-on experience with AI models to expand on his limited knowledge.
- Completing his AI-based university project.
- Understanding AI model customisation.

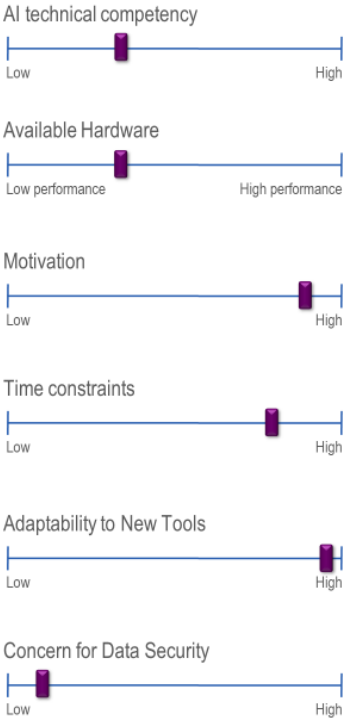
Goals

- To successfully incorporate AI into his university project.
- To experiment with various AI models (NLP, Vision).
- To gain practical experience by configuring models for his use case.

Pain points

- Lack of formal teaching in AI, causing gaps in knowledge, particularly implementation.
- Difficulty with accessing affordable AI tools.
- Time constraints due to project deadlines.

Behaviours and Attributes



ThoughtWorks

Figure C.1: Persona: Mark

Small Law Firm Owner

"AI could be the key to improving our firm's efficiency. But I am concerned about security"

Alan owns a small law firm and has read about some of the amazing opportunities AI presents. He wants to investigate if AI can be used to help his employees quickly search through old highly sensitive cases. Alan is hesitant to outsource to external companies due to data privacy concerns. He is looking for an inhouse solution that allows him to use AI in combination with his company's internal data. He has put aside a reasonable budget to pay for hardware, but wants to ensure he is using the right AI model for the job.

Name Alan Johnson
Type User
Role Small business owner

Motivations

- Wants to improve efficiency in tasks like document searching
- Focused on keeping client data private.
- Wants to stay competitive by adopting modern technology.

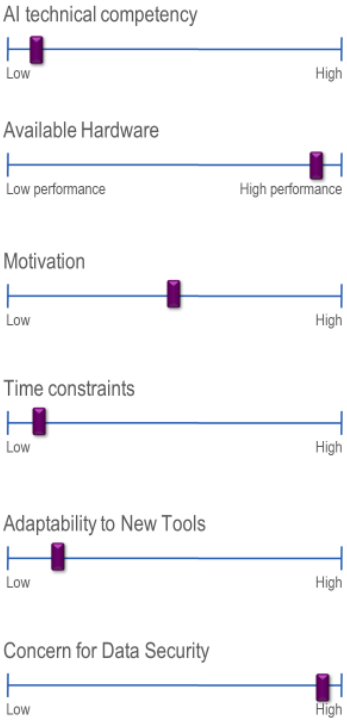
Goals

- Aims to implement AI to assist with legal research and document management.
- Wants a secure, in-house AI system.
- Aims to reduce routine tasks and boost productivity.

Pain points

- Concerned about privacy risks with external data processing.
- Finds selecting and using AI tools difficult due to limited knowledge.
- Wants to ensure any AI investment is cost-effective and integrates well.

Behaviours and Attributes



ThoughtWorks®

Figure C.2: Persona: Alan

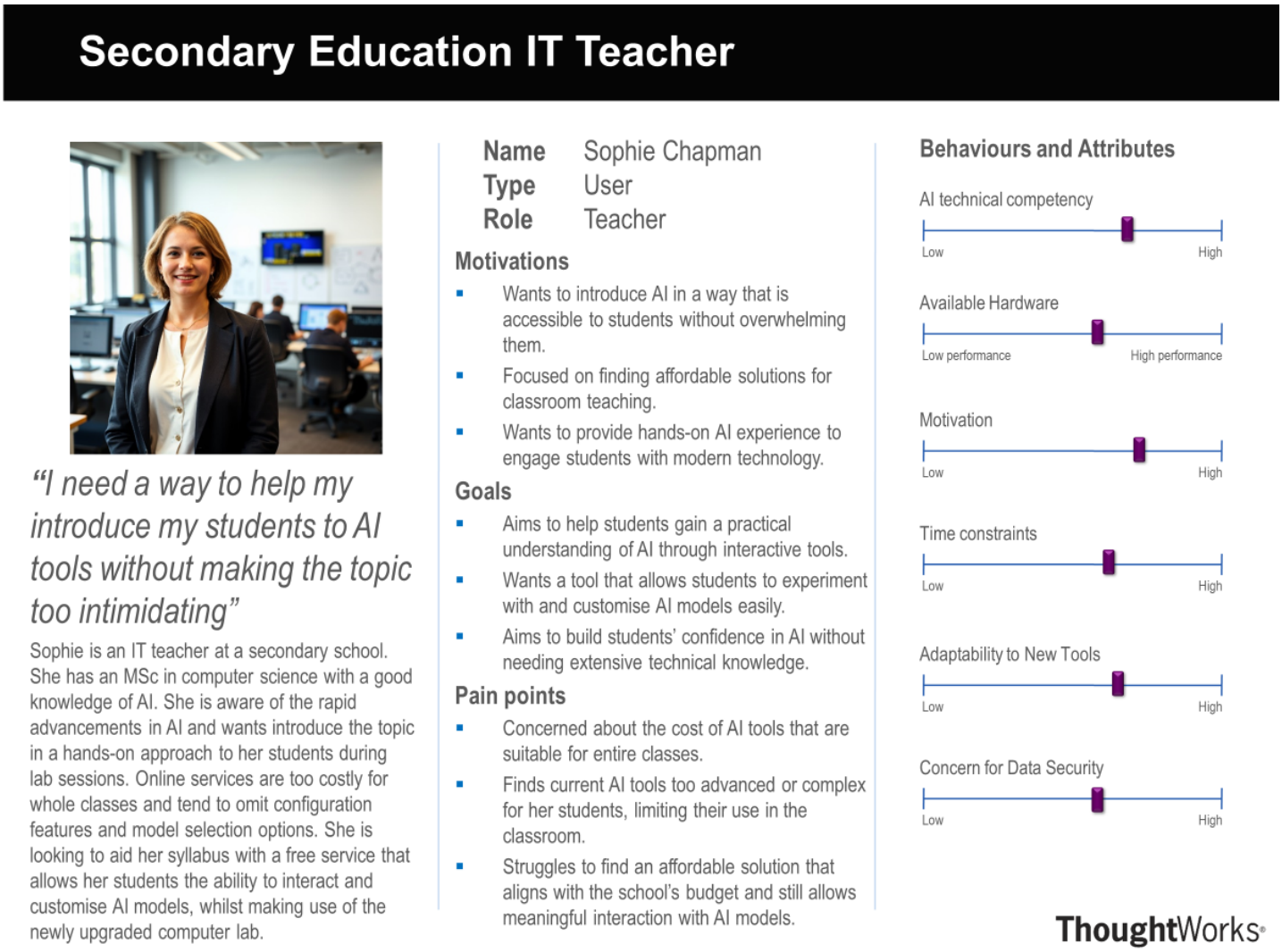


Figure C.3: Persona: Sophie

C.1.2 Use Case Resources

ID	Name	Brief Description of Use Case and Interaction
U1	ViewModelIndex	The user views the available models in the model index. This is the main use case for browsing models.
U2	SearchModelIndex	The user optionally searches for specific models before viewing the model index. This extends the ViewModelIndex use case.
U3	SaveModel	The user saves a selected model (online or offline) to their library. This use case includes interactions with external services such as IBM Watson (for online models) and Offline Model Sources (for offline models).
U4	Download Model	When saving an offline model, the system communicates with Offline Model Sources to download the model. This use case is included in SaveModel.
U5	GetWatsonData	When saving an online model, the system communicates with IBM Watson to retrieve the model. This use case is included in SaveModel.

ID	Name	Brief Description of Use Case and Interaction
U6	ViewSavedModels	The user views models that have already been saved to their local library. This is a direct interaction initiated by the user.
U7	SearchSavedModels	The user optionally searches for saved models before viewing them. This extends the ViewSavedModels use case.
U8	ViewModelInfo	The user views detailed information about a selected model after browsing or searching the model index. This extends ViewModelIndex and SearchSavedModels.
U9	DeleteSavedModel	The user deletes a model that has been previously saved in their library. This is a direct interaction initiated by the user.
U10	LoadModel	The user loads a previously saved model for inference or configuration. This use case includes the option to view hardware usage and run inference on the loaded model.
U11	ViewHardwareUsage	After loading a model, the user optionally views the system's hardware usage while running the model. This extends the LoadModel use case.
U12	InferenceModel	The user runs inference on a loaded model with custom input data. This use case includes LoadModel.
U13	StopModel	The user stops the loaded model. This use case includes LoadModel, as a model must be loaded to be stopped.
U14	ViewPlaygrounds	The user views available playgrounds for model testing and interaction. This is a direct interaction initiated by the user.
U15	SearchPlaygrounds	The user optionally searches through available playgrounds before viewing them. This extends the ViewPlaygrounds use case.
U16	CreatePlayground	The user creates a new playground after viewing available playgrounds. This extends the ViewPlaygrounds use case.
U17	ConfigureChain	The user configures a chain of models within an existing playground. This is a direct interaction that can occur at any time after a playground is created, so it is not directly dependent on CreatePlayground.
U18	LoadChain	The user loads a configured chain of models. This is a direct interaction initiated by the user.
U19	InferenceChain	The user runs inference on a loaded model chain. This use case includes LoadChain.
U20	StopChain	The user stops the loaded model chain. This use case includes LoadChain, as a chain must be loaded to be stopped.
U21	DeletePlayground	The user deletes an existing playground. This is a direct interaction initiated by the user.

ID	Name	Brief Description of Use Case and Interaction
U22	APIAccess	The user interacts with models or chains via an external API. This use case may extend LoadModel, StopModel, InferenceModel, LoadChain, StopChain, ConfigureChain and InferenceChain.

Table C.1: Use Case listing

C.1.3 Team Requirements Table

ID	Requirement	Priority
FR-G1	The App will feature a searchable index of offline and online AI/ML models, allowing users to search by model type, application area, and offline/online status.	Must Have
FR-G2	The App will feature the ability to save selected models from the index to a user-specific library.	Must Have
FR-G3	The App will feature the ability to delete models from the user-specific library.	Must Have
FR-G4	The App will feature the ability to load and unload models saved in the library.	Must Have
FR-G5	The App will feature the ability to have multiple models loaded simultaneously.	Must Have
FR-G6	The App will allow users to inference custom inputs using loaded models.	Must Have
FR-G7	The App will feature tools for fine-tuning and/or configuring saved models where relevant.	Must Have
FR-G8	The App will feature API access to saved models in order to enable external application integration and testing using load, unload and inference.	Must Have
FR-G9	The App will allow users to create playgrounds that can hold multiple models from the library.	Must Have
FR-G10	The App will allow users to delete playgrounds.	Must Have
FR-G11	The App will feature the ability to save chain configurations using both offline and online models together to create customised solutions, within a playground.	Must Have
FR-G12	The App will allow users to load and unload chain configurations into system memory.	Must Have
FR-G13	The App will feature the ability to inference configured model chains on custom inputs.	Must Have
FR-G14	The App will feature API Access to saved chain configurations in order to enable external application integration and testing, using load, unload and inference.	Must Have
FR-G15	The app will feature the ability to display the model information, including model specifications, ratings, and source.	Should Have
FR-G16	The App will feature the ability to view performance metrics for running offline models.	Should Have

FR-G17	The App will feature the ability to test models saved in the library on predefined test datasets.	Won't Have
FR-G18	The App will feature the ability to offload tasks to the Watson AI online services based on performance metrics.	Won't Have

Table C.2: General Functional Requirements

C.1.4 Individual Requirements Table

ID	Requirement	Priority
IR-1	The App will feature a selection of popular open-source offline text-gen AI models within a model index.	Must Have
IR-2	The App will feature extensive configuration options for text-gen models, including prompt tuning and parameter settings.	Must Have
IR-3	The App will feature optimisation, such as quantisation support, for text-gen models.	Must Have
IR-4	The App will feature chatbot capabilities for offline text-gen models, such as chat history and UI representation.	Must Have
IR-5	The App will feature RAG integration to allow for querying of user data.	Should Have
IR-6	The App will allow users to fine-tune text-gen models on custom training data.	Won't Have

Table C.3: Individual Requirements

C.2 Non-Functional Requirements Table

ID	Requirement	Priority
NFR-1	The App will feature a user-friendly, intuitive user interface.	Must Have
NFR-2	The App will be designed to allow easy updates to the model index.	Must Have
NFR-3	The App will be designed to allow for new model sources to be easily implemented, e.g., OpenAI, TensorFlow Hub, etc.	Must Have
NFR-4	The App will not send personal data to online services unless explicitly specified.	Must Have
NFR-5	The App will feature extensive error handling and logging.	Must Have
NFR-6	The App will feature a maintainable code base, with a focus on modularity and future feature expandability.	Must Have
NFR-7	The App will feature encryption for user-provided data.	Won't Have

Table C.4: Non-Functional Requirements

C.3 HCI Resources

C.3.1 Paper Sketches and descriptions

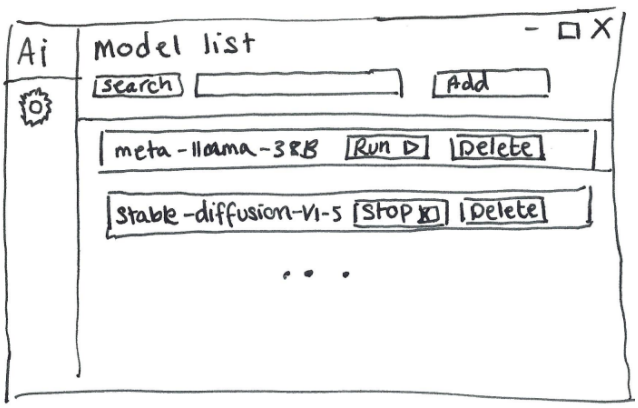


Figure C.4: Library

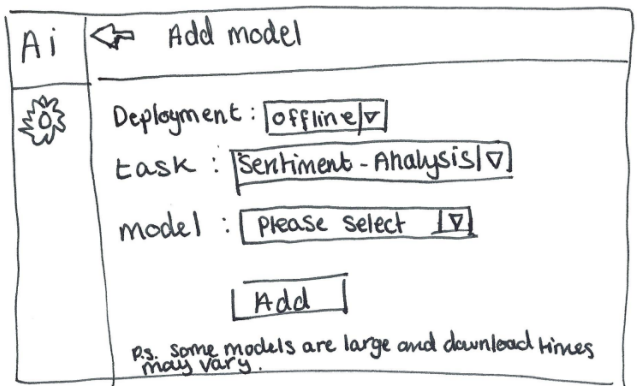


Figure C.5: Add Model From Library

The initial paper sketches did not include a dedicated AI Index. Instead, they featured a library page (**Figure C.4**), which allowed users to search through their list of downloaded models, run or stop them, and delete them. The AI Index functionality was captured through an ”Add Model” button, which opened a new page where users could add models using a progressive filtering system, as shown in **Figure C.5**.

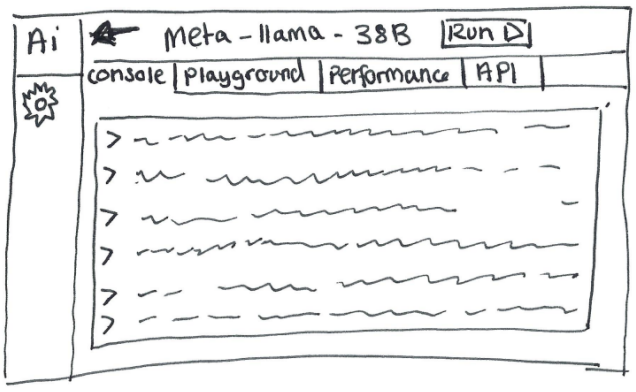


Figure C.6: Model View From Library

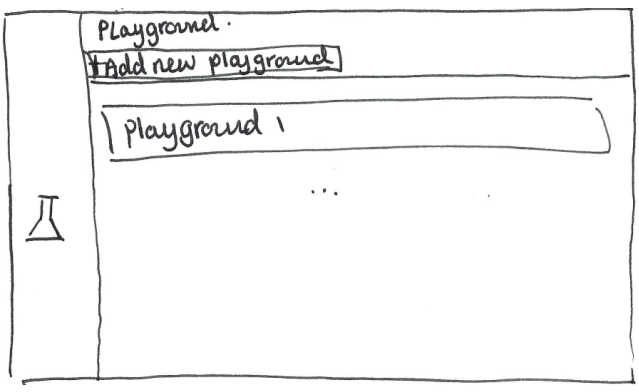


Figure C.7: List Of Playgrounds

Figure C.6 represents the general view when a user selects a model from their library, with a top bar providing access to model-specific features. A notable element here is the inclusion of the playground in the top navigation bar. This aspect of the UI underwent several iterations, ultimately being replaced by an inference tab, with the playground moved to a separate section, as shown in **Figure C.7**. Although this sketch provides a rough interpretation of the playground feature, it highlights the playground list view, which was retained throughout the design process.

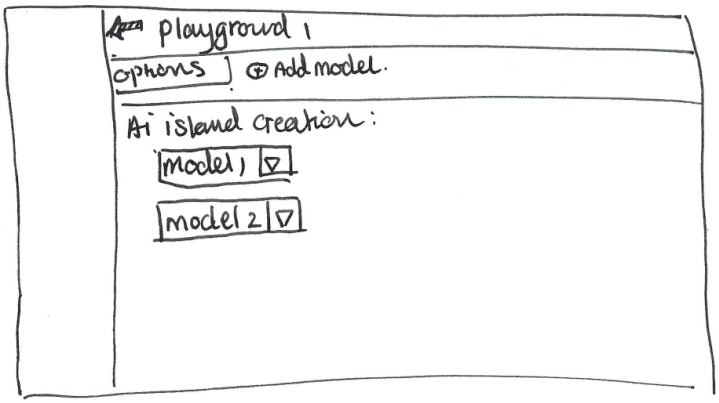


Figure C.8: Playground View From Playground List

Figure C.8 illustrates an early draft of the interface users would see when clicking on a playground from **Figure C.7**. The term ”AI Island Creation” in this sketch was later renamed to ”Chain Order” in the final application to better reflect its function.

C.3.2 Figma Prototype and Descriptions

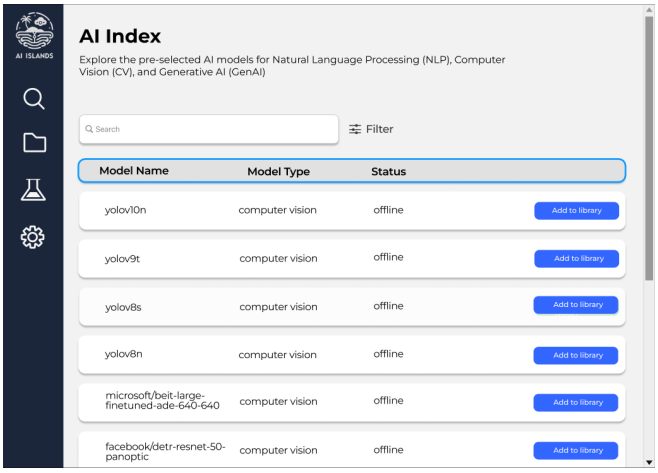


Figure C.9: AI Index

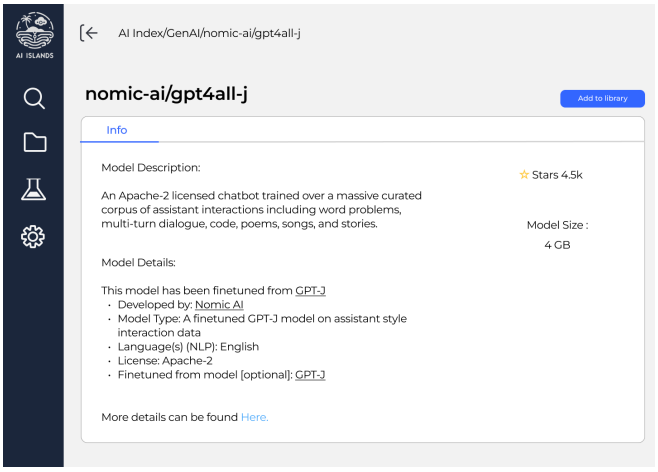


Figure C.10: Browse Model Info

One such example is the introduction of the AI Index, which replaces the previous implementation and provides a more user-friendly browsing experience. This is achieved through a clearer separation of concerns, as shown in **Figure C.9**, with **Figure C.10** offering a more intuitive presentation of model information.

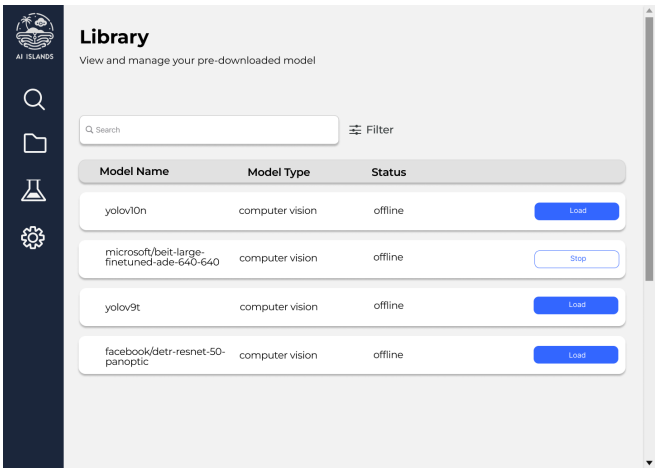


Figure C.11: Library

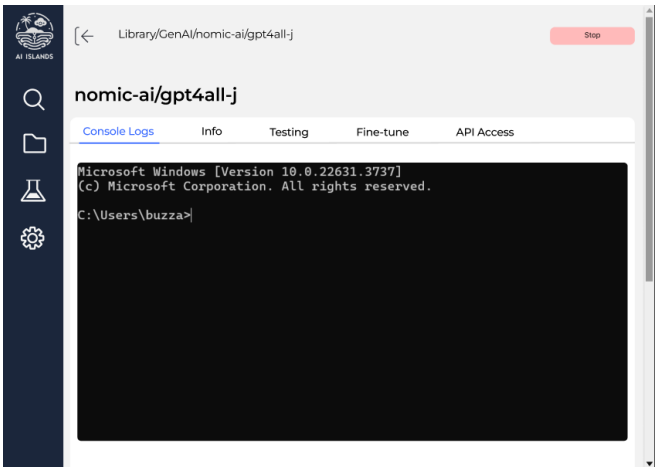


Figure C.12: Console View

Figure C.11 demonstrates the Library view, now focused exclusively on browsing and interacting with downloaded models. **Figure C.12** builds upon the earlier sketch (**Figure C.6**) and provides further context for the development of the tab system.

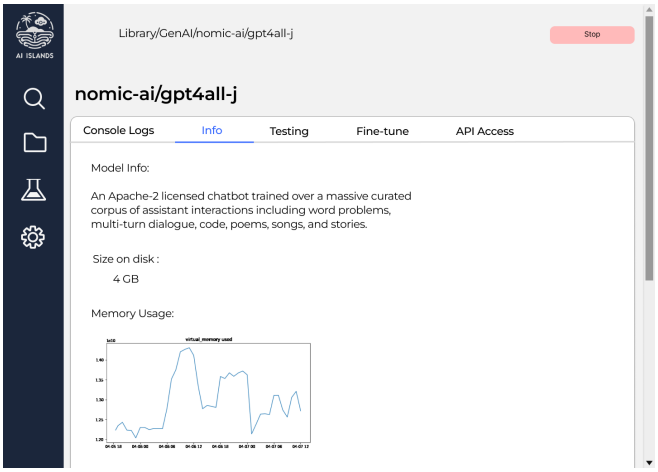


Figure C.13: Model View Info

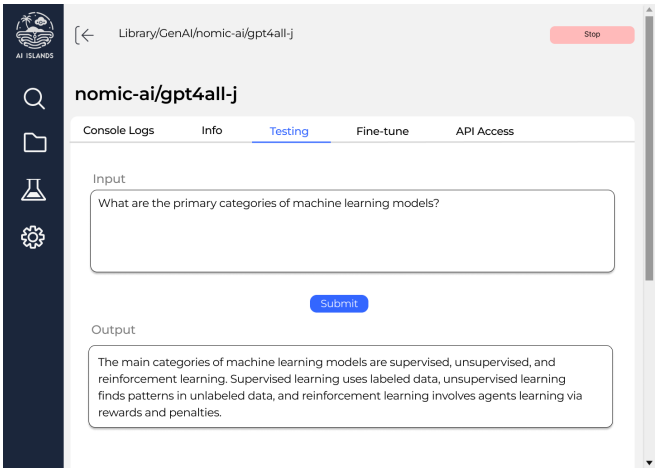


Figure C.14: Model View Testing

Figures C.13, C.14, and C.15 add more depth to the functionality of the tabs as well, outlining how key features of the AI Islands application, such as inferencing and API integration, will be accessed.

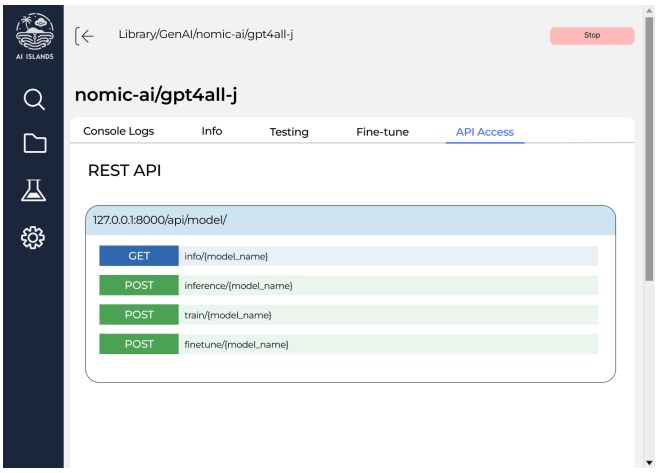


Figure C.15: Model View API Access

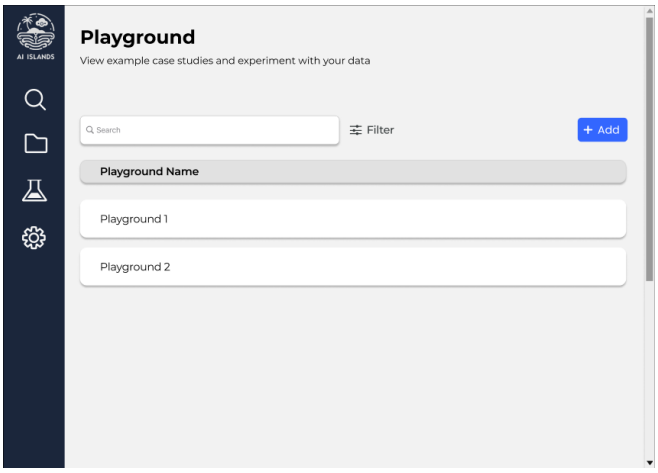


Figure C.16: Playground List

Figure C.16 shows further refinement of the paper sketch of the playground list view seen in **Figure C.7**, with an emphasis on a cleaner, more separated interface.

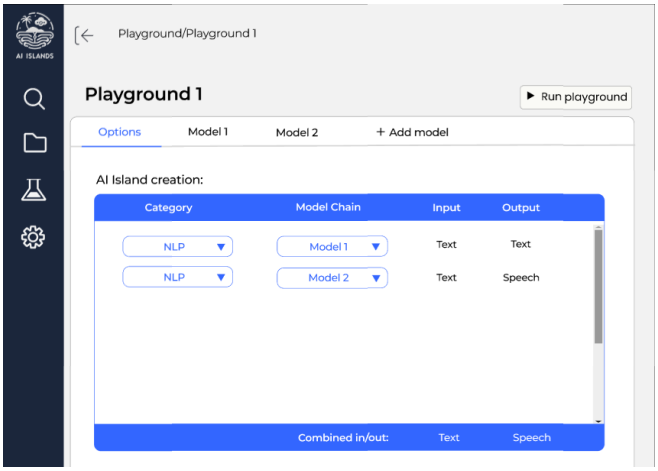


Figure C.17: Chain Config

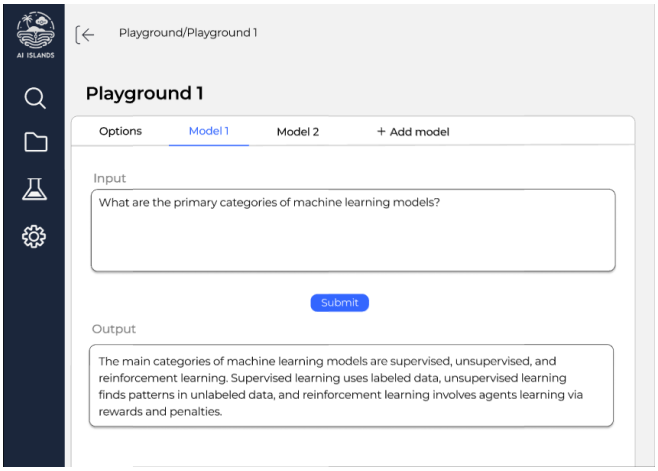


Figure C.18: Playground Inference

Finally, **Figure C.17** offers an iteration on the playground view design in **Figure C.8**, improving layout clarity and including relevant model information, while **Figure C.18** illustrates how individual models can be interacted with directly from the playground for testing purposes.

C.4 Research Resources

C.4.1 Similar Application Examples

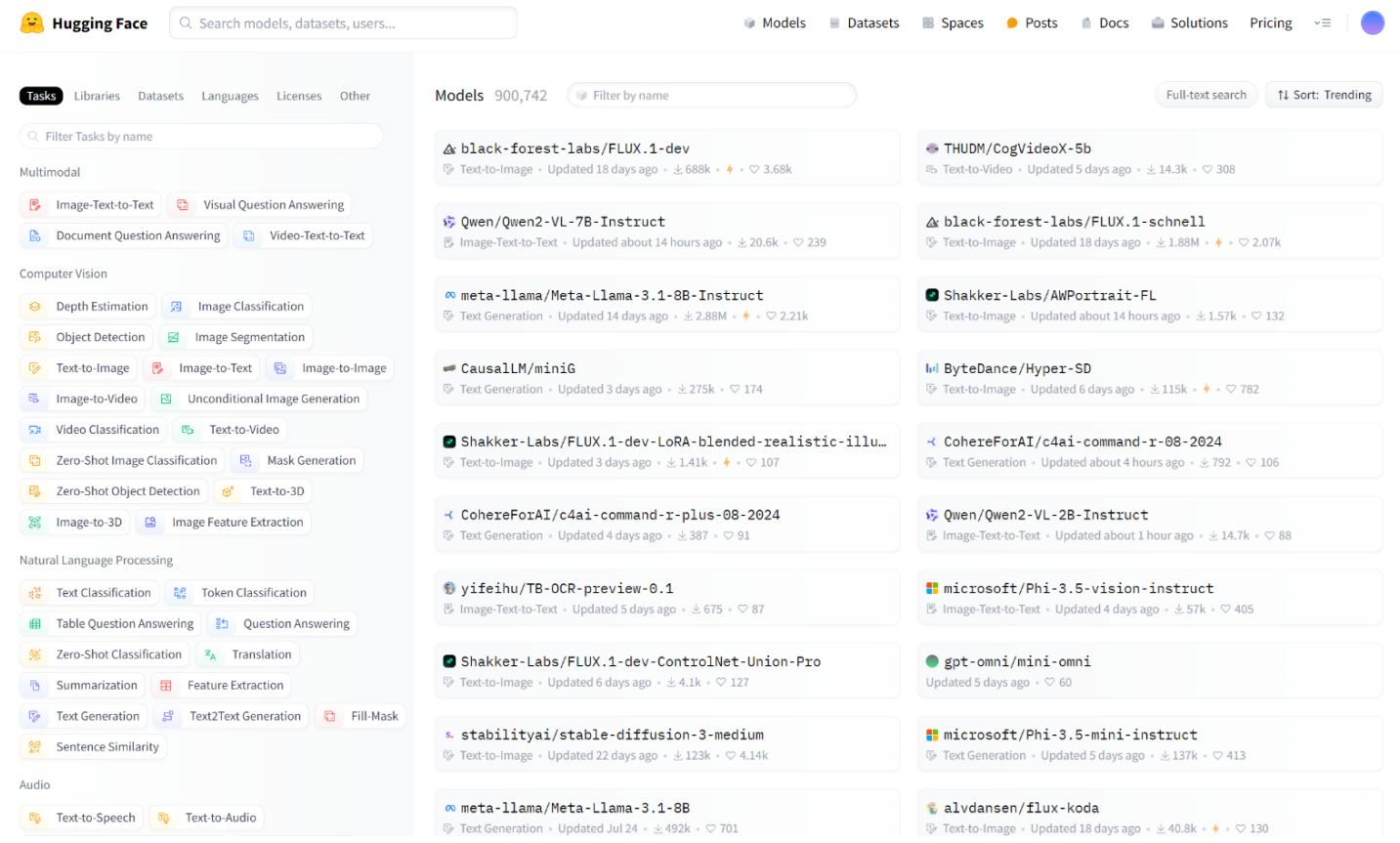


Figure C.19: Hugging Face model Index page [2]

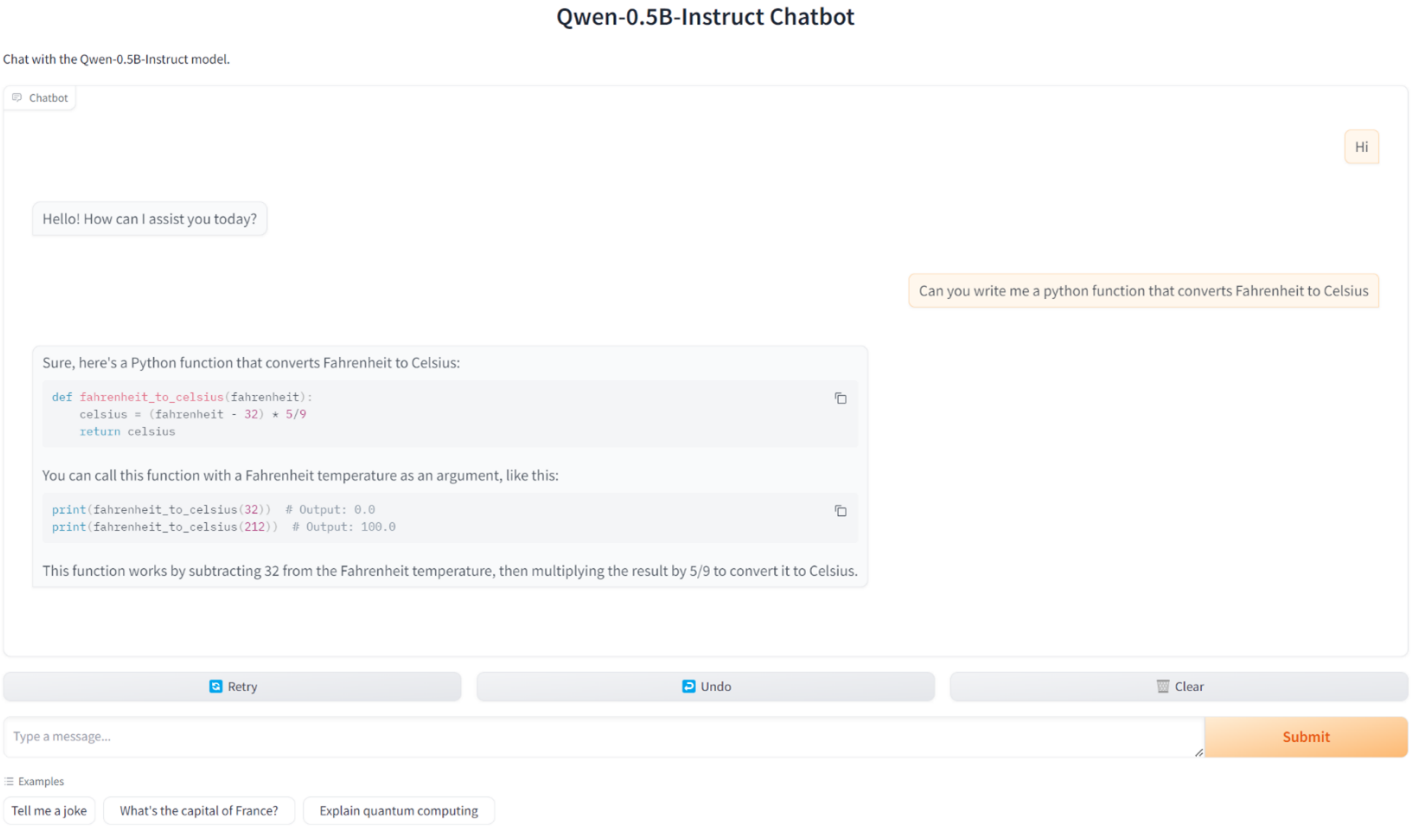


Figure C.20: A gradio deployment using a chatbot interface with the Qwen-0.5B-Instruct Chatbot [3]

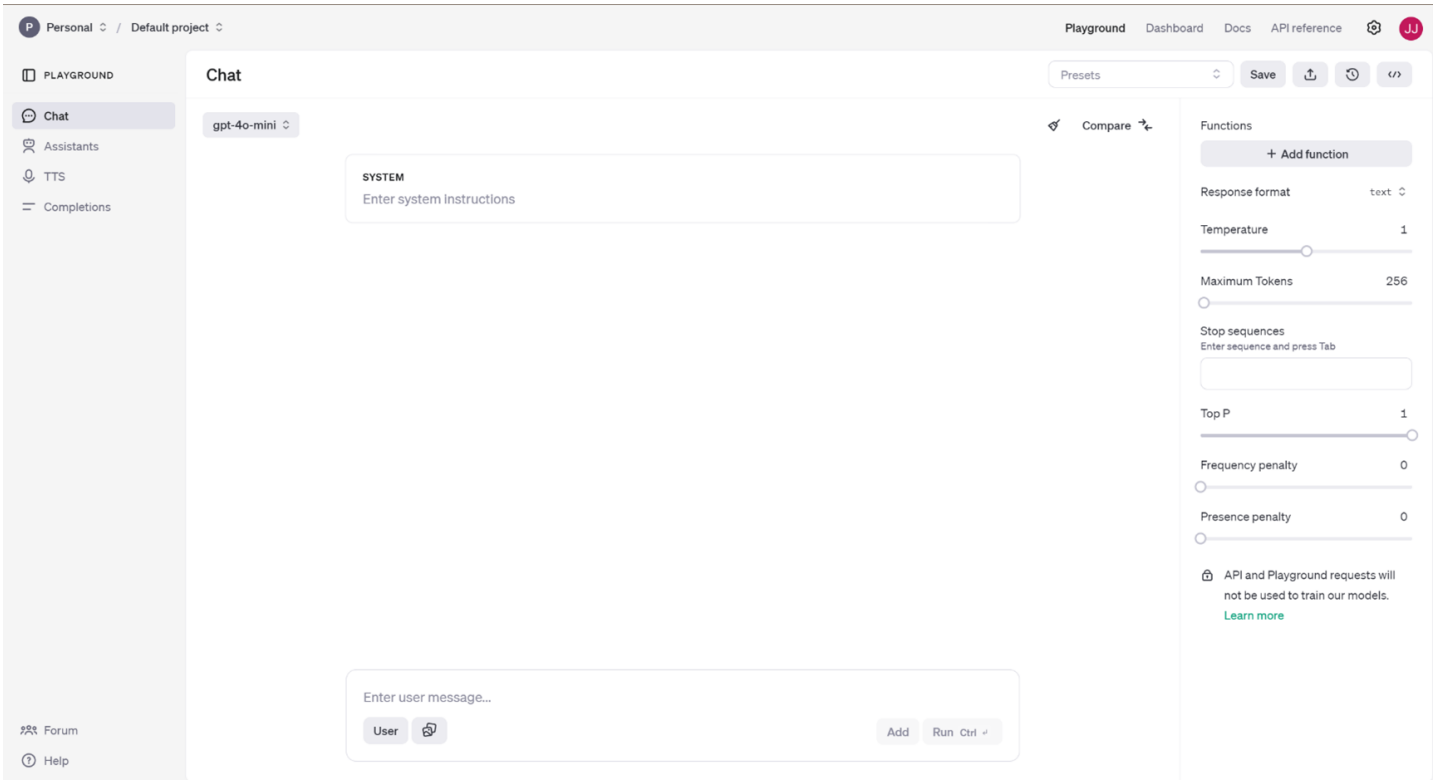


Figure C.21: OpenAI playground interface featuring config options on right hand side [4]

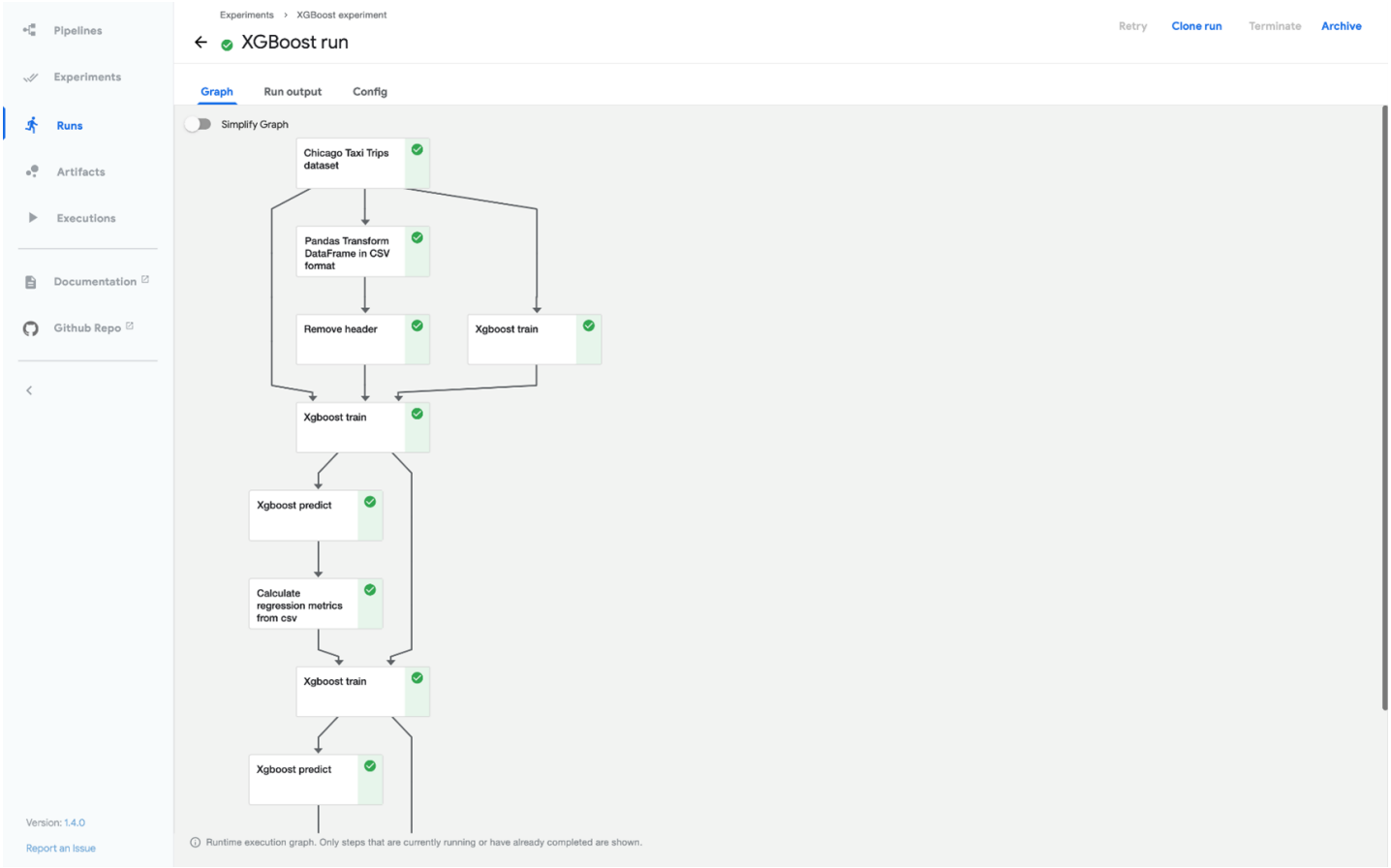


Figure C.22: Kubeflow interface demonstrating a machine learning based workflow [**?**]

C.5 System Design Resources

C.5.1 Initial class brainstorming

This Diagram represents the intial class brainstorming for the Model entity classes and the control class ModelControl, highlighting the first stage in class structure design.

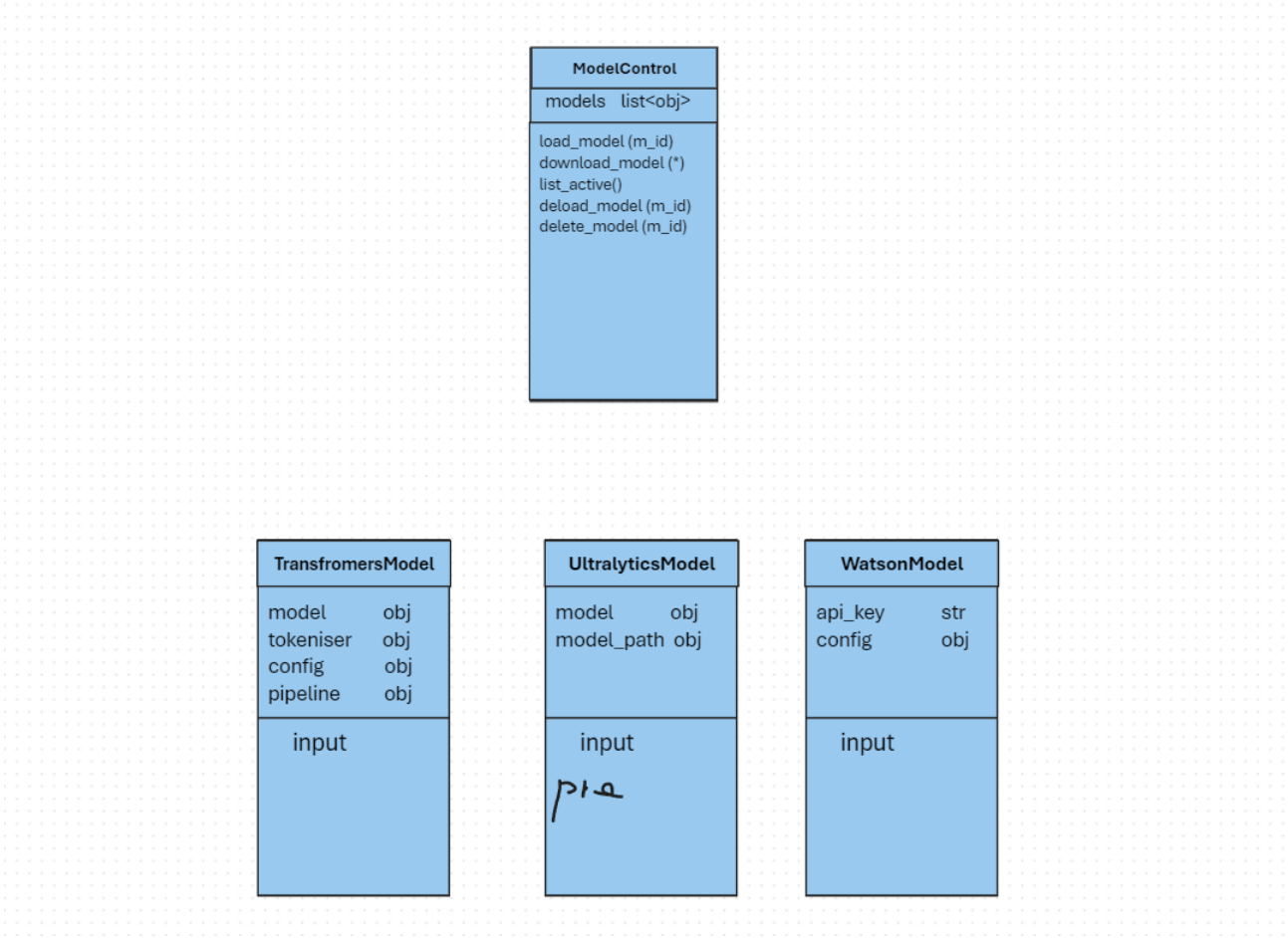


Figure C.23: Initial Class Brainstorming

C.5.2 Detailed Class diagram

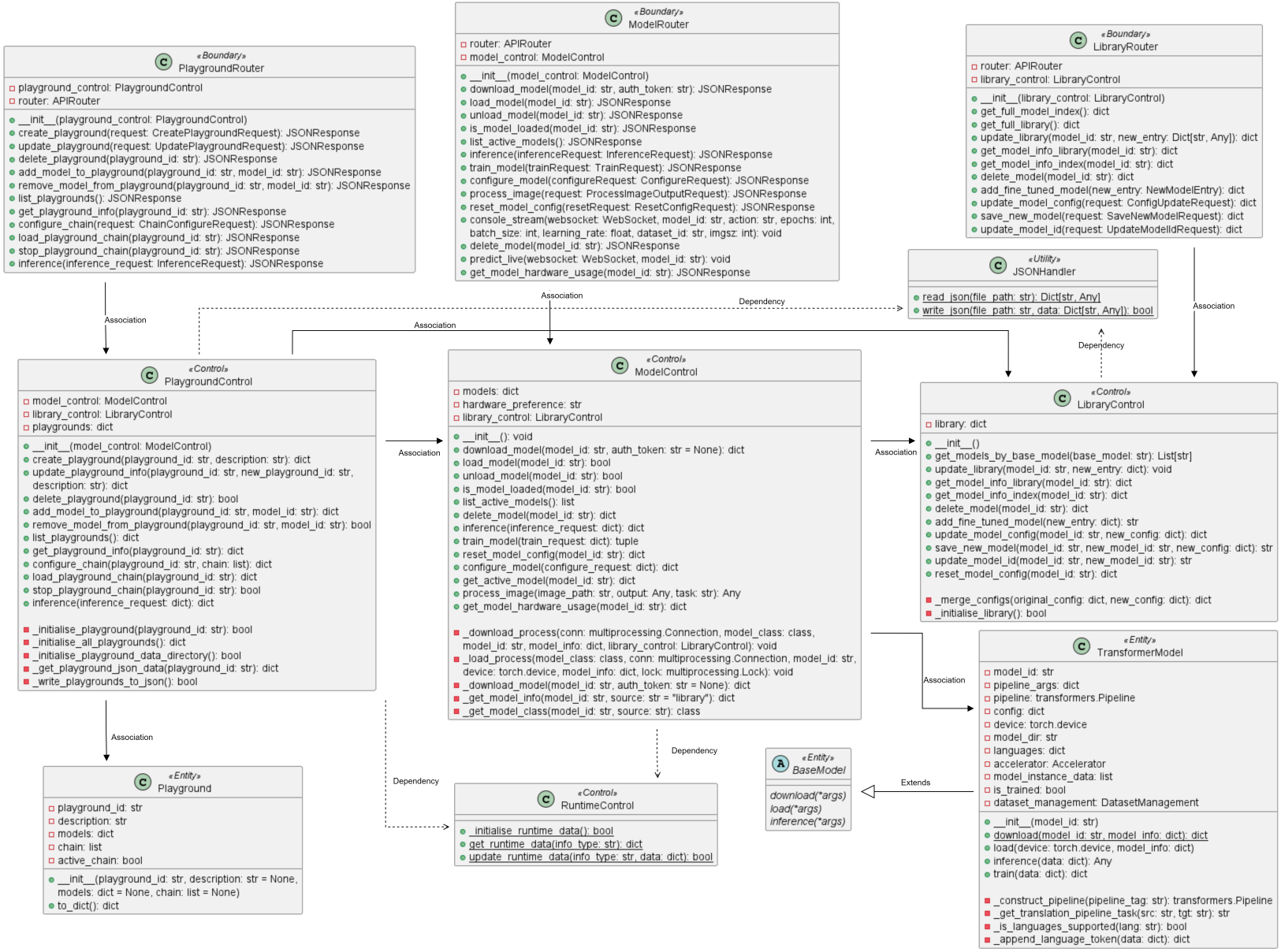


Figure C.24: Detailed Backend Class Diagram

C.5.3 Json Listings

```
{
   "model_id": "Unique identifier and name of the model",
   "is_online": "Boolean indicating if the model is from an online or offline source",
   "model_source": "String specifying the source of the model",
   "model_class": "String indicating the entity class used to instantiate the model",
   "mapping": "Dictionary detailing the input and output types of the model",
   "tags": "Array of strings representing relevant tags for the model",
   "pipeline_tag": "String indicating the task type of the model",
   "model_desc": "String providing a description of the model",
   "model_card_url": "String URL linking to the model's card or documentation",
   "requirements": "Dictionary of information required prior to loading the model",
   "config": "Dictionary containing base configuration options for the model"
}
```

Listing C.1: Minimum required attributes for a model entity

```
{
   "model_id_1": { /* model attributes minus model_id */ },
   "model_id_2": { /* model attributes minus model_id */ },
   "model_id_3": { /* model attributes minus model_id */ },
   "etc"
}
```

Listing C.2: 'model_index.json' Structure

```
{
   "model_id": {
      /* base attributes from model_index.json */
      "auth_token": "String authentication token for model loading if required",
      "base_model": "String model_id of the original model to enable model customisation",
      "dir": "String cache directory for loading downloaded models into memory",
      "is_customised": "Boolean indicating whether the model has been customised"
   }
}
```

Listing C.3: 'library.json' Structure

```
{
   "playground_id_1": {
      "description": "String description of the playground",
      "models": {
         "model_id_1": {
            "input": "String representing the model's input data type",
            "output": "String representing the model's output data type",
            "pipeline_tag": "String indicating the task type of the model",
            "is_online": "Boolean indicating if the model is from an online or offline source"
         },
         "etc"
      },
      "chain": ["model_id_1", "model_id_2", "etc"]
   }
}
```

Listing C.4: 'playground.json' Structure

```
{
   "playground": {
      "model_id_1": ["playground_id_1", "playground_id_2", "etc"],
      "model_id_2": ["playground_id_3", "etc"]
   }
}
```

Listing C.5: 'runtime_data.json' Structure

C.6 Implementation Resources

C.6.1 Folder structure

Figure C.25 represents a selected view of the backend folder structure, displaying the key folders and files nested within. While the description bellow it provides an extensive understanding of each of the folders responsibilities.

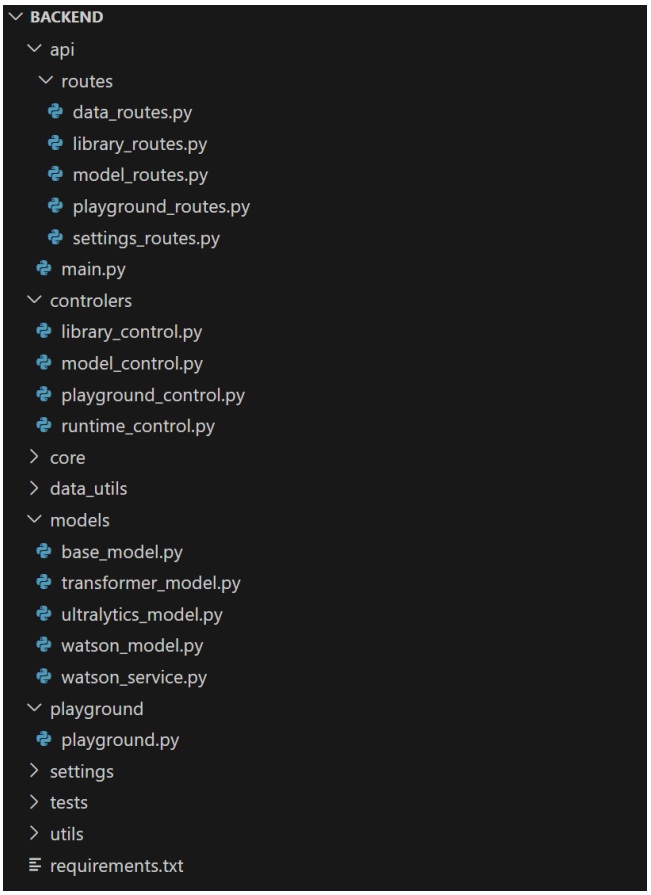


Figure C.25: Backend folder structure

The api folder serves as the main entry point, containing main.py and a routes subfolder, which holds the Router classes in separate files, as depicted in Figure 5.2.

The controlers folder contains various control classes, each in its own file, as outlined in Figure 5.2. This includes classes such as ModelControl, PlaygroundControl, and LibraryControl.

The core folder holds files related to general application functions, such as exception handling and directory path management.

The data_utils folder contains utilities specifically related to data processing and manipulation, including the JsonHandler class, as shown in Figure C.24.

The models folder unifies the AI model classes based on the BaseModel class. It contains the implementations for all model classes from different sources, such as TransformerModel.

The playground folder is responsible for housing the Playground class. Although it is an equivalent entity class to the models, its distinct responsibility warranted its placement in a separate folder.

The settings folder contains model-specific settings, applied on a per-model basis.

Finally, the utils folder houses all other utility classes that are not directly involved in data processing or manipulation.

C.6.2 main.py

This represents a reduced view of the main.py file, focusing on important features like class instantiation, object passing and router initialisation.

```
# Class imports

app = FastAPI()

# Instantiate controls and set all models inactive in chain
model_control = ModelControl()
library_control = LibraryControl()
playground_control = PlaygroundControl(model_control)

# Initialise library and runtime data files if they do not exist
library_control._initialise_library()
RuntimeControl._initialise_runtime_data()

# Create router instances
model_router = ModelRouter(model_control)
data_router = DataRouter()
library_router = LibraryRouter(library_control)
settings_router = SettingsRouter()
playground_router = PlaygroundRouter(playground_control)

# Include routers
app.include_router(model_router.router, prefix="/model", tags=["model"])
app.include_router(data_router.router, prefix="/data", tags=["data"])
app.include_router(library_router.router, prefix="/library", tags=["library"])
app.include_router(settings_router.router, prefix="/settings", tags=["settings"])
app.include_router(playground_router.router, prefix="/playground", tags=["playground"])

# main loop
```

Listing C.6: Reduced main.py

C.6.3 Default Routing

The following listings represent the standard implementation of API routes for a large application using dependency injection.

```
from fastapi import FastAPI, Depends
from some_class import SomeClass
from routers import router1

app = FastAPI()

# Create an instance of SomeClass
my_class = SomeClass(name="My Awesome Service")

# Include routers, passing the instantiated object directly
app.include_router(router1, prefix="/router1", dependencies=[Depends(lambda: my_class)])
```

Listing C.7: main.py view of class injection to router

```python
from fastapi import APIRouter, Depends
from some_class import SomeClass  # import for type hinting

router1 = APIRouter()

# Define a route in this router that depends on MyService
@router1.get("/resource1")
async def get_resource1(service: SomeClass = Depends()):
    return {"service_name": service.method1()}

@router1.get("/resource2")
async def get_resource2(service: SomeClass = Depends()):
    return {"service_name": service.method2()}
```

Listing C.8: Repetition of injection in routes

C.6.4 Routing Implementation

The following listings depict the refined implementation for the routing classes.

```python
from fastapi import APIRouter, Query
from some_class import SomeClass

class SomeRouter:
    def __init__(self, some_class: SomeClass):
        self.router = APIRouter()
        self.some_class = some_class

    # Define routes
    self.router.add_api_route("/resource1", self.get_resource1, methods=["GET"])
    self.router.add_api_route("/resource2", self.get_resource2, methods=["GET"])

    async def get_resource1(self):
        return {"service_name": self.some_class.method1()}

    async def get_resource2(self):
        return {"service_name": self.some_class.method2()}
```

Listing C.9: Router Class Implementation

```python
from fastapi import FastAPI
from some_class import SomeClass

app = FastAPI()

# Instantiate classes
some_class = SomeClass()

# Create router instances
some_router = SomeRouter(some_class)

# Include routers
app.include_router(some_router.router, prefix="/some", tags=["someTag"])
```

Listing C.10: Adding Router Instance to main.py

C.6.5 Full API Routes Listing

This section will outline all Route class constructors and their featured API Routes relevant to this project report.

```python
class ModelRouter:
   def __init__(self, model_control: ModelControl):
      self.router = APIRouter()
      self.model_control = model_control

      # Routes
      self.router.add_api_route("/download-model", self.download_model, methods=["POST"])
      self.router.add_api_route("/load", self.load_model, methods=["POST"])
      self.router.add_api_route("/unload", self.unload_model, methods=["POST"])
      self.router.add_api_route("/is-model-loaded", self.is_model_loaded, methods=["GET"])
      self.router.add_api_route("/active", self.list_active_models, methods=["GET"])
      self.router.add_api_route("/inference", self.inference, methods=["POST"])
      self.router.add_api_route("/configure", self.configure_model, methods=["POST"])
      self.router.add_api_route("/reset-config", self.reset_model_config, methods=["POST"])
      self.router.add_api_route("/hardware-usage", self.get_model_hardware_usage,
      ↪  methods=["GET"])
      self.router.add_api_route("/delete-model", self.delete_model, methods=["DELETE"])
```

Listing C.11: ModelRoutes

```python
class LibraryRouter:
    def __init__(self, library_control: LibraryControl):
        self.router = APIRouter()
        self.library_control = library_control

        self.router.add_api_route("/get-full-model-index", self.get_full_model_index, methods=["GET"])
        self.router.add_api_route("/get-full-library", self.get_full_library, methods=["GET"])
        self.router.add_api_route("/get-model-info-library", self.get_model_info_library, methods=["GET"])
        self.router.add_api_route("/get-model-info-index", self.get_model_info_index, methods=["GET"])
```

Listing C.12: LibraryIRoutes

```python
class PlaygroundRouter:
    def __init__(self, playground_control: PlaygroundControl):
        self.playground_control = playground_control
        self.router = APIRouter()

        self.router.add_api_route("/create", self.create_playground, methods=["POST"])
        self.router.add_api_route("/update", self.update_playground, methods=["PUT"])
        self.router.add_api_route("/delete", self.delete_playground, methods=["DELETE"])
        self.router.add_api_route("/add-model", self.add_model_to_playground, methods=["POST"])
        self.router.add_api_route("/remove-model", self.remove_model_from_playground, methods=["POST"])
        self.router.add_api_route("/list", self.list_playgrounds, methods=["GET"])
        self.router.add_api_route("/info", self.get_playground_info, methods=["GET"])
        self.router.add_api_route("/configure-chain", self.configure_chain, methods=["POST"])
        self.router.add_api_route("/load-chain", self.load_playground_chain, methods=["POST"])
        self.router.add_api_route("/stop-chain", self.stop_playground_chain, methods=["POST"])
        self.router.add_api_route("/inference", self.inference, methods=["POST"])
```

Listing C.13: PlaygroundIRoutes

C.6.6 Process Handling

The following Listings display pseudo code implementations of the given methods, with comments to express irrelevant code.

```
def load_model(self, model_id: str):
    # Functionality related to extracting required information based on the model_id


   # Child process spawning with relevant information
   lock = multiprocessing.Lock()
   parent_conn, child_conn = multiprocessing.Pipe()
   process = multiprocessing.Process(target=self._load_process, args=(model_class,
   ↪  child_conn, lock, "Other relevant data"))
   process.start()

   response = parent_conn.recv()

   # Error handling and result return
```

Listing C.14: load_model Pseudo code

```
@staticmethod
def _load_process(model_class, conn, model_id, device, model_info, lock):

   # instantiate the model class with model_id
   model = model_class(model_id=model_id)
   model.load(device=device, model_info=model_info)
   conn.send("Model loaded")

   # A loop to keep the process alive
   while True:
      req = conn.recv()
      # Termination check
      if req == "terminate":
         conn.send("Terminating")
         break
      elif #Task check:
         with lock:
            # Inference request using Lock
            if req["task"] == "inference":
               result = model.inference(req["data"])
               conn.send(result)
```

Listing C.15: _load_process Pseudo code

```
@staticmethod
self.models[model_id] = {'process': process, 'conn': parent_conn, 'model': model_class,
↪  'pid': process.pid}
```

Listing C.16: self.models dictionary

C.6.7 System Usage Gathering

```
def get_model_hardware_usage(self, model_id: str):
    if model_id in self.models:
        pid = self.models[model_id]['pid']

        process = psutil.Process(pid)
        cpu_percent = process.cpu_percent(interval=1)
        memory_info = process.memory_info()
        memory_percent = process.memory_percent()

        if torch.cuda.is_available():
            gpus = GPUtil.getGPUs()
            if gpus:
                gpu = gpus[0]
                gpu_usage = gpu.memoryUsed
                gpu_total = gpu.memoryTotal
                gpu_percent = (gpu_usage / gpu_total) * 100 if gpu_total > 0 else 0

        # Error handling and formatting
        result = {
            'cpu_percent': round(cpu_percent, 2),
            'memory_used_mb': round(memory_info.rss / (1024 * 1024), 2),
            'memory_percent': round(memory_percent, 2),
            'gpu_memory_used_mb': round(gpu_usage, 2) if gpu_usage is not None else None,
            'gpu_memory_percent': round(gpu_percent, 2) if gpu_percent is not None else None,
        }
```

Listing C.17: get_model_hardware_usage Pseudocode

C.6.8 TrasnformerModel and data customisation Listings

```
def download(model_id: str, model_info: dict):

    # Gathering the required classes and their configs from the model_info

    for class_type, class_name in required_classes.items():

        # dynamically import the class from transformers library, ex: AutoModelForSeq2SeqLM
        class_ = getattr(transformers, class_name)

        # Dynamically get the config for the class_type
        obj_config = config.get(f'{class_type}_config', {})

        if auth_token:
            obj_config['use_auth_token'] = auth_token

        # download the class object from huggingface transformers library
        _obj = class_.from_pretrained(
            model_id,
            cache_dir=model_dir,
            **obj_config
        )
```

Listing C.18: Pseudocode version of dynamic imports

```
{
"meta-llama/Meta-Llama-3.1-8B-Instruct": {
    " Existing Attributes as detailed in the System design chapter"
    "..."
    "requirements": {
      "required_classes": {
        "model": "AutoModelForCausalLM",
        "tokenizer": "AutoTokenizer"
      },
      "requires_auth": true
    },
    "config": {
      "chat_history": false,
      "model_config": {
        "torch_dtype": "bfloat16"
      },
      "tokenizer_config": {},
      "processor_config": {},
      "pipeline_config": {
        "max_length": 512,
        "max_new_tokens": 1000,
        "num_beams": 2,
        "use_cache": true,
        "temperature": 0.6,
        "top_k": 40,
        "top_p": 0.92,
        "repetition_penalty": 1.2,
        "length_penalty": 1.2
      },
      "device_config": {
        "device": "cuda"
      },
      "quantization_config": {
        "current_mode": "4-bit"
      },
      "quantization_config_options": {
        "4-bit": {
          "load_in_4bit": true,
          "bnb_4bit_use_double_quant": true,
          "bnb_4bit_quant_type": "nf4",
          "bnb_4bit_compute_dtype": "bfloat16"
        },
        "8-bit": {
          "load_in_8bit": true
        },
        "bfloat16": {}
      },
      "system_prompt": {
        "role": "system",
        "content": "You are a helpful assistant."
      },
      "user_prompt": {
        "role": "user",
        "content": "[USER]"
      },
      "assistant_prompt": {
        "role": "assistant",
        "content": "[ASSISTANT]"
      },
      "example_conversation": [
        {
          "role": "user",
          "content": "How are you?"
        },
        {
          "role": "assistant",
          "content": "I'm good, thanks! What can I help you with?"
        }
      ],
      "rag_settings": {
        "use_dataset": false,
        "dataset_name": null,
        "similarity_threshold": 0.5,
        "use_chunking": false
      }
    }
  }
}
```


Listing C.19: Json For Custom Data Implementation Of Text-Generation Models

C.6.9 Quantisation and optimisation Pseudo code

```
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline

from accelerate import Accelerator

from transformers import BitsAndBytesConfig

accelerator = Accelerator()

bnb_config = BitsAndBytesConfig("4 bit or 8 bit quantisation settings")

tokenizer = AutoTokenizer.from_pretrained( "model_name", cache_dir)

model = AutoModelForCausalLM.from_pretrained(model_name, cache_dir, bnb_config)

model = accelerator.prepare(model)

text_generator = pipeline(task_type, model, tokenizer)

text_generator = accelerator.prepare(text_generator)

output = text_generator(input_text, pipeline_args)
```

Listing C.20: Python pseduo code for Integration of Accelerate and BitsAndBytes

C.7 Testing Resources

C.7.1 Unit Tests

```python
def test_download_success(model_info_index):
    expected_model_dir = os.path.join('data', 'downloads', 'transformers', "Qwen/Qwen2-0.5B-Instruct")

    with patch('backend.models.transformer_model.os.path.exists', return_value=False) as mock_exists, \
        patch('backend.models.transformer_model.os.makedirs') as mock_makedirs, \
        patch('backend.models.transformer_model.transformers.AutoModelForCausalLM.from_pretrained') as mock_model, \
        patch('backend.models.transformer_model.transformers.AutoTokenizer.from_pretrained') as mock_tokenizer:

        mock_model.return_value = MagicMock()
        mock_tokenizer.return_value = MagicMock()

        updated_model_info = TransformerModel.download("Qwen/Qwen2-0.5B-Instruct", model_info_index)

        # Assertions
        mock_exists.assert_called_once_with(expected_model_dir)
        mock_makedirs.assert_called_once_with(expected_model_dir, exist_ok=True)
        mock_model.assert_called_once_with(
            "Qwen/Qwen2-0.5B-Instruct",
            cache_dir=expected_model_dir,
            **model_info_index['config']['model_config']
        )
        mock_tokenizer.assert_called_once_with(
            "Qwen/Qwen2-0.5B-Instruct",
            cache_dir=expected_model_dir,
            **model_info_index['config']['tokenizer_config']
        )
        assert updated_model_info["base_model"] == "Qwen/Qwen2-0.5B-Instruct"
        assert updated_model_info["dir"] == expected_model_dir
        assert updated_model_info["is_customised"] is False
        assert "config" in updated_model_info
```

Listing C.21: Successful Download Test

```python
@pytest.fixture
def transformer_model(model_info_library):
    model = TransformerModel(model_id="Qwen/Qwen2-0.5B-Instruct")
    model.config = copy.deepcopy(model_info_library['config'])
    model.pipeline = MagicMock()
    model.pipeline.task = model_info_library.get('pipeline_tag')
    model.model_instance_data = []
    return model

def test_inference_text_generation_without_chat_history(transformer_model):
    test_input = "Hello, how are you?"
    test_data = {
        "payload": test_input,
        "pipeline_config": {"max_length": 100}
    }

    mock_output = [{"generated_text": [{"content": "I'm doing well, thank you for asking!"}]}]
    transformer_model.pipeline.return_value = mock_output

    with patch.object(transformer_model, 'config', {
        'user_prompt': {"role": "user", "content": "[USER]"},
        'chat_history': False
    }):
        output = transformer_model.inference(test_data)

    expected_input = [{"role": "user", "content": "Hello, how are you?"}]
    transformer_model.pipeline.assert_called_once_with(expected_input, max_length=100)
    assert output == "I'm doing well, thank you for asking!"
```

Listing C.22: Successful inference Test

C.7.2 Integration Tests

```python
class MockTransformerModel(TransformerModel):
    def __init__(self, model_id: str):
        super().__init__(model_id)
        self.model_id = model_id
        self.config = {}
        self.model_dir = 'mock/dir'
        self.is_trained = False
        self.pipeline_tag = 'text-generation'
        self.device = 'cpu'
        self.pipeline = MagicMock()
        self.pipeline.task = self.pipeline_tag
        self.pipeline.side_effect = self._pipeline_return_value

    def _pipeline_return_value(self, *args, **kwargs):

        if args[0][0]['content'] == "model error test":
            raise ModelError("Inference failed")
        else:
            return [{"generated_text":[{"role":"assistant","content":"I'm an AI assistant. How
            ↪  can I help you today?"}]}]

```

Listing C.23: MocktransformerModel

```
@pytest.fixture
def model_control(model_info_library):
    with patch('backend.controlers.model_control.ModelControl._get_model_class',
    ↪  return_value=MockTransformerModel), \

        patch('backend.controlers.model_control.LibraryControl.
        get_model_info_library', return_value=model_info_library), \

        patch('backend.controlers.model_control.LibraryControl.
        get_model_info_index', return_value=model_info_library), \

        patch('backend.settings.settings_service.SettingsService
        .get_hardware_preference', return_value='cpu'):

        model_control = ModelControl()
        model_control.load_model(model_info_library['base_model'])
        yield model_control
        # Clean up after the test
        model_control.unload_model(model_info_library['base_model'])
```

Listing C.24: Patching MockTransformerModel before loading

```
def test_inference_success(client, model_info_library):
    model_id = model_info_library['base_model']
    inference_data = "Hello, how are you?"
    expected_output =  "I'm an AI assistant. How can I help you today?"

    response = client.post("/model/inference", json={"model_id": model_id, "data":
    ↪  {"payload":inference_data}})
    print(response.json())
    assert response.status_code == 200
    assert response.json()["data"] == expected_output
```

Listing C.25: Inference Success Integration test

C.7.3 Test Teardown

```
# Store original data
original_model_index = JSONHandler.read_json(MODEL_INDEX_PATH)
original_downloaded_models = JSONHandler.read_json(DOWNLOADED_MODELS_PATH)
original_playground_data = JSONHandler.read_json(PLAYGROUND_JSON_PATH)

@pytest.fixture(autouse=True)
def reset_application_data():
    # Setup: Nothing to do, original data is already stored
    yield
    # Teardown: Reset all data to original state
    JSONHandler.write_json(MODEL_INDEX_PATH, copy.deepcopy(original_model_index))
    JSONHandler.write_json(DOWNLOADED_MODELS_PATH,
    ↪  copy.deepcopy(original_downloaded_models))
    JSONHandler.write_json(PLAYGROUND_JSON_PATH, copy.deepcopy(original_playground_data))
```

Listing C.26: Test Teardown

C.7.4 Test-Group UAT Members

Participant	Position	Background	Relevancy
P1	Senior Software Engineer	P1 has over 30 years of experience in the software engineering industry.	P1 is interested in providing hands-on AI tools to interns under their supervision without consuming company resources.
P2	Data Analyst Consultant	P2 has been working as a technical consultant for 2 years and holds a degree in physics.	P2 aims to incorporate offline AI models to assist with data analysis but has concerns regarding their company's data security.
P3	Illustrator/Graphic Designer	P3 has over 30 years of experience in their field but has limited experience with AI models, mainly using tools like ChatGPT and MidJourney.	P3 is concerned about the rapid development of AI and its potential impact on job security, and wants to better understand how the technology works.

Table C.5: UAT Participants

C.7.5 UAT Task List

Task	Requirement	Description	P1	P2	P3
UAT-01	FR-G1, IR-1	Search and filter the AI Index for an offline text-generation model.	Pass	Pass	Pass
UAT-02	FR-G15	Click on a model from the AI Index and identify its information.	Pass	Pass	Pass
UAT-03	FR-G2	Download the 'meta-llama/Meta-Llama-3.1-8B-Instruct' model and view it in the Library.	Pass	Pass	Pass
UAT-04	FR-G4	Load the 'meta-llama/Meta-Llama-3.1-8B-Instruct' model from the Library.	Pass	Pass	Pass
UAT-05	FR-G16	Click on the loaded model and identify the hardware usage information.	Pass	Pass	Pass
UAT-06	FR-G8	Navigate to the API Access page for the loaded model and identify the requests.	Pass	Pass	Fail
UAT-07	FR-G7, IR-2	Navigate to the Configuration Tab, enable chat history, and save the configuration.	Pass	Pass	Pass
UAT-08	IR-3, FR-G7, NRF-7, IR-2	Change the quantisation mode to 4-bit in the Configuration page and save the settings.	Pass	Pass	Pass
UAT-09	IR-4	Select the provided supplementary dataset for RAG in the Configuration page and save the settings.	Pass	Pass	Fail
UAT-10	FR-G6, IR-4	Navigate to the Inference Tab and start a chat with a successful reply from the model.	Pass	Pass	Pass
UAT-11	IR-5, FR-G6	Ask the model a question based on supplementary data and receive a correct response.	Pass	Pass	Fail
UAT-12	FR-G4, FR-G7	Restore configurations to default and unload the model from the library.	Pass	Pass	Pass
UAT-13	FR-G10, FR-G9	Navigate to the Playground, delete the existing playground, and add a new one.	Pass	Pass	Pass
UAT-14	FR-G9	Add the 'meta-llama/Meta-Llama-3.1-8B-Instruct' and an offline text-to-speech model to the Playground.	Pass	Pass	Pass

UAT-15	FR-G11	Save a chain featuring the ‘meta-llama/Meta-Llama-3.1-8B-Instruct‘ model followed by the TTS model.	Pass	Pass	Pass
UAT-16	FR-G14	Locate the Playground API Access page and identify the requests.	Pass	Pass	Fail
UAT-17	FR-G12, FR-G5	Navigate to the Playground Inference Tab and load the chain.	Pass	Pass	Pass
UAT-18	FR-G13	Submit an input to the loaded chain and verify the result.	Pass	Pass	Pass
UAT-19	FR-G12, FR-G10	Unload the chain and delete the created Playground.	Pass	Pass	Pass
UAT-20	FR-G3	Find the ‘meta-llama/Meta-Llama-3.1-8B-Instruct‘ in the Library and delete it.	Pass	Pass	Pass

Table C.6: UAT Tasks and Results

C.7.6 Test-Group UAT Feedback

Participant	Feedback
P1	P1 found the UI intuitive and responsive but suggested that additional visual feedback could be helpful when waiting for a model chain to load or for inference results.
P2	P2 appreciated the UI but suggested that chain configuration could be made more intuitive, such as by using draggable models that could be manually connected via lines.
P3	P3 was initially intimidated by the task list but was positively surprised by how easy the UI made the process. Their primary feedback was to provide more descriptive detail on the Configuration and API Access pages to assist less experienced users.

Table C.7: UAT Participant Feedback