AI Islands: Integration with Watsonx

FWDM7

Internal Supervisor: Professor Graham Roberts

COMP0073 MSc Computer Science Project
MSc Computer Science
September 23, 2024

Department of Computer Science
University College London

Abstract

AI Islands is an innovative project designed to unify and integrate various isolated artificial intelligence (AI) models, both online and offline, into a cohesive, user-driven platform. This desktop application, developed using .NET MAUI with a Python backend powered by FastAPI, functions as a versatile playground where users can seamlessly combine, experiment with, and customise AI models according to their specific needs. The primary focus of this specific sub-project is to integrate IBM's watsonx models into the AI Islands app, enabling users to tap into IBM's AI capabilities for tasks such as inference, configuration, and model chaining.

By linking the user's IBM watson cloud account to the app, AI Islands allows for flexible offloading of computation tasks to the cloud, offering an alternative to running models locally. The app simplifies the use of IBM's suite of AI services, ranging from foundation models to their natural language processing tools. The project also explores the enhancement of watsonx models through programmatic use, including the implementation of a Retrieval-Augmented Generation (RAG) system, where dataset processing is either done locally or via a watsonx embedder, thus reinforcing the *Islands* concept. Datasets can serve as the contextual base for Large Language Models (LLMs) within the application, which also features chatbot capabilities for interacting with these LLMs.

AI Islands exemplifies the vision of creating a flexible, customisable platform where AI models from multiple sources can interact, providing users with the freedom to design tailored AI systems that meet their unique requirements. Through this integration of IBM's advanced AI tools, the app delivers a unified and user-friendly experience, making cutting-edge AI more accessible and adaptable for experimentation and real-world application.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Project Overview

In this project, we will analyse the key concepts and technical requirements behind an *AI Islands* themed application. Our focus will be on exploring programmatic access to AI resources through a locally run application, which is central to the integration of IBM watsonx. Additionally, we will explore how to enhance AI models by customising model configurations, incorporating chatbot functionality, designing and utilising a Retrieval-Augmented Generation (RAG) system, and implementing model chaining for more advanced workflows.

1.1.1 The AI Islands Theme

The *Island* theme represents the concept of distinct environments, or 'islands', where different types of AI models reside. Each island hosts a specific model type, allowing users to manage and operate these models independently. However, within the unified AI Islands application, all models share common capabilities such as inferencing and configuration, enabling users to interact with them in a consistent manner. Despite their isolation, these islands are interconnected, allowing seamless integration and interaction between models. This theme emphasises modularity and flexibility, encouraging users to experiment with and combine various AI technologies within a cohesive system.

1.1.2 IBM Watsonx

Watsonx is IBM's AI platform that contains many AI and data analysis tools. With watsonx being an isolated AI system to the IBM Cloud, we can bring it into a unified environment, making it easier to create, manage, and deploy a range of AI models from different sources that are able to communicate and collaborate together. This can lead to a more generalised application for the user.

1.2 Problem Statement

The rise of AI is a relatively new technology and we are already seeing the rapid improvements within the AI field, such as with OpenAI's powerful ChatGPT. However, there are still issues that need to be addressed concerning cost and efficiency, security, ease of access, and online dependency.

Cost and Efficiency: There is still a demand for low-cost (or free), yet accurate, AI tools. As computer hardware is constantly improving, there is now a focus on running offline AI models locally. In addition to this, paid AI tools such as IBM's watsonx have limited exposure for users who cannot afford the cost, meaning less usage overall.

Security: There are concerns with using sensitive data for online model training and prompting. For example, if you wanted to query medical or company accounts data, or fine-tune a model with the data, the professional would ideally prefer a locally run offline AI tool.

Ease of Access: Beginners face difficulty accessing AI and machine learning resources. Additionally, AI tools can be overwhelming for beginners and customising or integrating advanced models can be challenging, especially without a user-friendly interface. For example, a user may want to chain AI models together to produce their desired output. This would require sufficient coding knowledge in order to accomplish this.

Online Dependency: There is a heavy reliance on online and cloud-based AI models; there is limiting functionality where users opt for no internet use or have no internet access.

1.3 Problem Solution

The proposed solution to the identified problems as described in section 1.2 is the *AI Islands Desktop Application*; a unified platform for managing, configuring, and running different types of AI models in a user-friendly interface. We can resolve the identified problems as follows:

Cost and Efficiency Solution: We can include both locally run models and cloud-based run models within the same application. This will allow users to choose free offline models or choose an IBM cloud-based model if they wish. This would tackle any costing issues and IBM's depreciating AI service exposure.

Security Solution: We can use locally run AI models such as offline large language models (LLMs) along with either a RAG system or fine-tuning capability. This will mean users can contextualise their prompts with datasets, with no risk of security issues.

Ease of Access Solution: A desktop application with a user-friendly interface designed for beginners will address the issue of accessibility. AI Islands will feature a comprehensive model index that offers a wide range of AI models along with their descriptions and specifications, allowing users to easily modify individual model configurations with guided assistance. Additionally, users have the ability to perform model chaining within a dynamic, playground-like environment, where they can easily fine-tune models or execute Retrieval-Augmented Generation (RAG) through a learnable and intuitive UI.

Online Dependency Solution: AI Islands will cover a range of offline models, thus nullifying the online dependency issue.

1.4 Project Goals

The primary goal of the AI Islands project is to create a comprehensive, robust, and user-friendly platform for managing AI models, with seamless integration to IBM's watsonx at its core. The platform will offer a wide array of tools for model customisation, allowing users to configure models for specific tasks and execute efficient inferencing. It will incorporate support for advanced functionalities such as RAG, enabling users to enhance model performance by providing contextual data. Furthermore, the platform is designed to be scalable, enabling it to support future advancements in AI by integrating new models and tools from isolated systems. This ensures that users can easily access new and developing technologies, allowing for continuous expansion and adaptation.

1.5 Project Structure

This project follows a structured development approach, divided into Research & Design, Project Development, and Testing & Reporting phases.

During the Research & Design phase (Weeks 1-4), the focus was on requirements gathering, stakeholder meetings, research, and prototype designing. Key tasks included experimenting with watsonx AI tools and IBM cloud services, as well as creating experimental app prototypes.

The Project Development phase (Weeks 5-13) consisted of backend and frontend implementation, with tasks such as developing the core functionality, integrating a RAG system, and enabling model chaining. Backend testing was conducted during this phase to ensure the applications's reliability.

Finally, the Testing & Reporting phase (Weeks 14-16) centered around testing the app's performance and preparing documentation. The project concluded with client handover and presentations.

Please see the Gantt Chart in **Figure A.1** for more detail on project structure.

1.6 Reporting Overview

In this project, we will primarily explore the various ways to integrate IBM's watsonx into AI Islands. We will begin by providing the necessary background on AI models and IBM's watsonx platform, followed by a requirements and use case analysis, and then outline the general structure of AI Islands. We will then move onto the more technical aspects, focusing on IBM Cloud programmatic access and the main integration of watsonx with AI Islands. This will be followed by analysis of the RAG system and how we can enhance watsonx AI models through their programmatic use. This report will finish with application testing, followed by a conclusion.

References to the appendix will be made throughout this report. Additionally, supplementary materials are provided alongside the report, which the reader is encouraged to explore, yet optional. These materials will be referenced throughout to offer additional context and support the discussions presented. The project demonstration video can be found in **Section 10.6**.

Chapter 2

Background and Introduction to Watsonx

In this chapter, we will cover the necessary background for this project, introducing various types of AI models and how they can be run. We will focus particularly on IBM's watsonx AI models and services, offering a high-level overview based on research and investigation of their resources, without delving into technical details.

2.1 Introduction To AI Models

There are many different types of AI models. To the average person, large language models (LLMs) are probably the most well-known type, especially since the introduction of OpenAI's ChatGPT. However, there are many other types that fall into the category of natural language processing and computer vision. AI Islands' model index consists of models from these categories.

In fact, AI Islands' model selection covers both locally-run and cloud-run AI models, which we will refer to as *offline* and *online* in this report. There are advantages and disadvantages to both types. For offline models, model files are downloaded to the users' computer, which can require a lot of memory depending on the model size. The user would also require a powerful computer processor in order to run these models, especially if they want good performance and quick response times.

For online models, a download is not necessary since all processing occurs on a companies' cloud. This means quick response times, regardless of the users' computer specifications. However, for cloud-based models, there is usually a cost incurred on the user; generally through a subscription to use a company service. In this project report, we will be focusing particularly on online cloud-based AI models, more specifically the IBM Cloud.

2.2 IBM Cloud Platform

The IBM Cloud platform provides a variety of services, primarily for businesses to build, run, and manage applications. It offers infrastructure options such as virtual servers and storage solutions, like the IBM Cloud Object Storage [1], that allow companies to scale their operations without the need for physical hardware. The platform also includes development tools such as the IBM Kubernetes Service [2] for managing containerised applications. Additionally, IBM Cloud provides ready-to-use software, such as IBM watsonx Assistant [3] for building AI-powered chatbots and IBM Maximo [4] for enterprise asset management.

2.3 What Is Watsonx?

Watsonx is IBM's AI and data platform that provides a range of tools and services. It fully supports the entire AI life-cycle, from data preparation to model deployment and governance. It consists of foundation models [5], machine learning tools, and data processing, all within a cloud-based environment. Watsonx is designed for organisations to build, deploy, and manage AI models to enhance their business goals. For convenience, IBM have divided watsonx into three components which we will briefly cover; watsonx.ai [6], watsonx.data [7], and watsonx.governance [8].

2.3.1 Watsonx.ai

Watsonx.ai is the AI environment within watsonx, focused on model training, deployment, and fine-tuning. It provides access to pre-trained foundation models and tools for creating custom AI models tailored to specific business needs.

Example: A business could use watsonx.ai to fine-tune a large language model for customer service, making it more responsive to industry-specific queries.

2.3.2 Watsonx.data

Watsonx.data is a data management and analytics component that integrates with IBM's data fabric, allowing seamless access and analysis of data across different environments. It is designed for scalability and performance, enabling businesses to manage large datasets efficiently.

Example: A company can use watsonx.data to integrate and analyse customer data from multiple sources to gain insights into customer behavior and improve decision-making.

2.3.3 Watsonx.governance

Watsonx.governance provides tools for managing the ethical and responsible use of AI. It ensures that AI models are transparent, fair, and compliant with regulations, addressing concerns around AI bias and accountability.

Example: An organisation might use watsonx.governance to monitor and audit AI models to ensure they meet ethical standards and avoid biased decision-making.

2.4 Watson Studio

Watson Studio is IBM's integrated environment for building, training, and deploying AI models [9], available through both *Cloud Pak for Data* and *Watsonx* [10] [11]. While both platforms support AI development, they emphasise different aspects of the AI and data life-cycle [12]. Different organisations have varying requirements, making one platform more suitable than the other depending on their specific needs [13] [14].

Cloud Pak for Data aligns more closely with the watsonx.data component, focusing on data governance, integration, and large-scale data management. In this context, Watson Studio is primarily used for deploying and managing AI models within data-centric workflows, where strong governance and data handling are a priority. This makes *Cloud Pak for Data* ideal for organisations that need comprehensive data management alongside their AI operations.

On the other hand, *Watsonx* is more focused on advanced AI life-cycle management, particularly in relation to the watsonx.ai component. In this context, Watson Studio is optimised for model customisation, fine-tuning, and large-scale deployment, using the powerful collection of foundation models for AI tasks. *Watsonx* is suited for organisations that prioritise AI development and scaling, offering tools for deploying advanced AI capabilities efficiently.

For this report, the emphasis will be on Watson Studio within the *Watsonx* platform, specifically its role in watsonx.ai for prompting foundation models, training, and deploying AI models in a scalable and adaptable environment. A visual representation is provided in Appendix C.4.

2.5 How Can Watsonx Be Integrated?

Integration with watsonx involves using IBM's AI services within the AI Islands application. This integration allows users to access watsonx services programmatically through the use of API keys [15], linked to their IBM cloud accounts. By doing so, users can interact with cloud-based AI services, such as foundation models, directly from the AI Islands interface. This capability enhances the app's functionality by enabling cloud-based inferencing, while ensuring secure and authenticated access to IBM's AI resources.

In practice, this integration allows users to incorporate watsonx models into their workflows without needing to manage local resources. By using these models, AI Islands can offload intensive tasks such as LLM inferencing to the cloud, providing a crucial alternative to users; especially for those who cannot use the hardware-constrained offline LLMs. Furthermore, this seamless integration reduces the need for the technical expertise required for using cloud-based products, such as Watson Studio, making the platform resources more accessible for users.

2.6 IBM Resource Availability

In this project, we will focus on resources that are available for a *lite* plan, as opposed to paid plans only. *Lite* plans cover usage of Watson Machine Learning's AI foundation and embedder models through Watson Studio, Natural Language Understanding (NLU), Text to Speech (TTS), and Speech to Text (STT). These resources are available through APIs, and are therefore included in AI Islands' model index. However, services such as Tuning Studio [16] and Watson Discovery [17] are not available on *lite* plans, but are particularly useful for fine-tuning and data searching respectively, and should be noted for potential future work in AI Islands, where paid accounts can be used for testing.

2.7 Similar Platform & API Analysis

According to our research, to fully cover a similar product analysis, we need to identify products that show two concepts; an AI index where users can experiment with inferencing different models independently or via model chaining, and the ability to use an API key for programmatic access to running models on a cloud-based platform. We will focus on OpenAI and LangChain.

2.7.1 OpenAI Platform

The OpenAI platform is a versatile web-based platform designed to run various models from text-generation to image-generation. The platform has an intuitive interface [18] where models can be inferenced, configured and fine-tuned, without the user requiring advanced technical skills, making it accessible to both developers and non-developers. The platform offers configuration changes on their playground page as shown below in Figure 2.1.

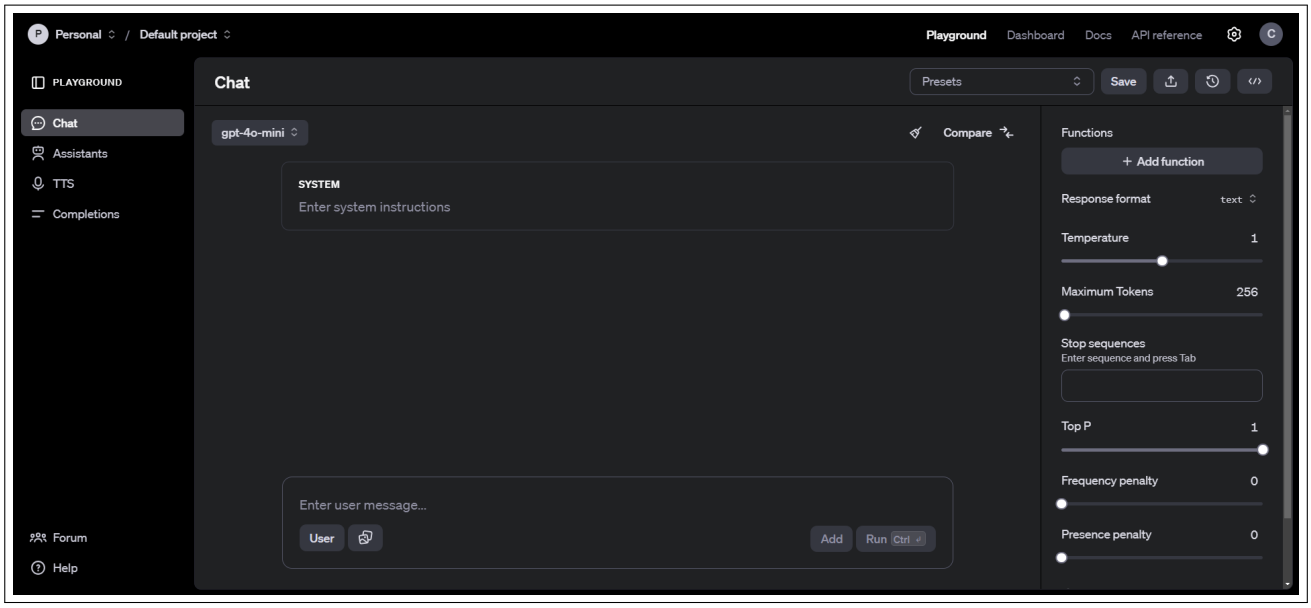


Figure 2.1: OpenAI Platform - Playground

This feature allows users to inference with adjustable settings such as temperature and tokens, while instantly observing the effects of their adjustments, making it easy to refine outputs. The simplicity and flexibility of this platform make it a valuable reference point for AI Islands, as it demonstrates how a user-friendly interface can support this kind of interaction with powerful AI models.

2.7.2 OpenAI GPT API

OpenAI GPT API is a very powerful tool for integrating LLMs into applications, requiring the user to enter their OpenAI API key in order to programmatically use OpenAI models, [19]. This is very similar to what is needed in AI Islands. The request is sent using the *OpenAI* package specific functions, and processed on the cloud, and the result is then returned to the local application. The basic syntax is shown in Listing 2.1.

```python
from openai import OpenAI

# Initialize the client
client = OpenAI(api_key='your-api-key')

# Create a chat completion
response = client.chat.completions.create(
    model="openai-model",
    messages=[{"role": "user", "content": "user-prompt"}]
)

# Print the response
print(response.choices[0].message.content)
```

Listing 2.1: Example of OpenAI API Key Integration in Python

The syntax here is very similar to how we set up and inference watsonx foundation models, as seen later in Listings 6.1 & 6.10.

2.7.3 LangChain

LangChain is a framework that simplifies integration of LLMs like OpenAI's GPT. It excels at prompt management and model chaining. LangChain's focus on API key-based access and cloud integration aligns closely with AI Islands, making it a relevant comparison for managing AI services programmatically. Developers can import LangChain libraries to achieve this [20]; see the example high-level concept in Listing 2.2.

```python
from langchain.llms import HuggingFaceLLM
from langchain.chains import LLMChain, SimpleSequentialChain
from langchain.prompts import PromptTemplate

# Define a model, prompt, and chain for the first model
model_1 = HuggingFaceLLM(model_name)
prompt_1 = PromptTemplate(input_variables, template)
chain_1 = LLMChain(model_1, prompt_1)

# Define a second model.
model_2 = HuggingFaceLLM(model_name)
prompt_2 = PromptTemplate(input_variables, template)
chain_2 = LLMChain(model_2, prompt_2)

# Continue defining models, prompts, and chains as needed...

# Combine the chains into a sequential process and run.
overall_chain = SimpleSequentialChain(chains=[chain_1, chain_2,...])
final_result = overall_chain.run()
```

Listing 2.2: Example of LangChain Integration in Python

2.8 Chapter Summary

In this chapter, we introduced the concepts and background necessary for understanding the AI Islands project, in particular with respect to watsonx integration. We began by exploring different types of AI models, primarily focusing on the difference between offline and online models. The advantages of using IBM's cloud-based models in AI Islands were discussed, highlighting how they enhance the application's scalability and provide seamless access to advanced AI services.

We then provided a detailed overview of IBM's watsonx platform, covering its main components (watsonx.ai, watsonx.data, and watsonx.governance) and how they support the full AI life-cycle. Next, we examined Watson Studio, a core tool within watsonx, comparing its usage in both *Cloud Pak for Data* and *Watsonx*, and focusing on its role in enabling AI model usage within AI Islands via programmatic access.

Finally, we explored the integration of watsonx into AI Islands, identifying which cloud-based AI services can be accessed programmatically, and conducted a comparative analysis of similar platforms (OpenAI and LangChain) with a particular focus on their API capabilities. These comparisons highlighted the key features of watsonx and its alignment with the goals of AI Islands.

Chapter 3

Requirements

In this chapter, we will introduce the main stakeholders for AI Islands. We will cover the requirements gathering process leading up to the MoSCoW requirements list, followed by the use cases and a use case diagram. Due to the nature of other projects being directly involved, the requirements are shown in terms of the application, with the appropriate abstraction to highlight this particular project.

3.1 Stakeholders

The AI Islands project involves key stakeholders including project sponsors Prof. Dean Mohamadally and Prof. John McNamara, and IBM technical advisor Chiranjib Das. Each stakeholder plays a role in guiding the project, with varying levels of influence and interest. Regular initial meetings and feedback had been established to ensure alignment with the project goals.

3.2 Requirement Elicitation Methods

To understand the business domain and client requirements for our application, we employed a combination of meetings, presentations, and thorough document analysis.

Stakeholder Meetings: We conducted targeted interviews and meetings with key project stakeholders to gain crucial insights, clarify objectives, and shape the product vision. These interactions were instrumental in aligning the project goals with stakeholder expectations.

IBM Technical Consultations: Meetings with IBM personnel provided some insights into the potential integration of watsonx with AI Islands. However, there were several uncertainties around using watsonx programmatically, making research and experimentation critical for the project's success. An experimental app, *The Watsonx Prototype* (discussed in Chapter 6), sparked further discussions on additional features to incorporate into AI Islands, such as watsonx embedder models.

Stakeholder Presentations: Presentations to IBM officials at their headquarters were key in gathering direct feedback from stakeholders who are deeply invested in the project. The feedback received during these sessions was integral to refining our approach and ensuring stakeholder buy-in.

Document Analysis: We performed in-depth analysis of relevant documentation, which allowed us to propose relevant features to the stakeholders. This approach helped to shift opinions on certain features, leading to adjustments in requirements that were more aligned with the project brief. As a result, meetings became more efficient and focused.

By combining these methods, we ensured a thorough understanding of the requirements of AI Islands, aligning the development process with the expectations of all stakeholders.

3.3 The Brief

The following project title and brief were provided for this project:

> *Title:* IBM-UCL AI Islands Theme: integration with watsonx
> *Brief:* Create functionality to allow integration between Island AI and watsonx
> - particularly the extension of the AI capability to watsonx.

The brief was discussed with the stakeholders (identified in section 3.1) which led to several iterative changes. The primary goal of this project is to integrate watsonx services into a local application that also incorporates models from other sources, such as Huggingface Transformers and Ultralytics. This integration aims to provide a seamless user experience, enabling various AI models to function and work together within a single system.

The AI *Islands* theme further solidifies this brief, representing the vision of uniting distinct AI models from isolated systems. The key challenge is finding an effective way to combine these models seamlessly and incorporate watsonx online services into the local application without compromising performance or usability. Conceptually, the project as a whole is essentially a collection of different types of AI models working together, as shown in the visual representation in Figure 3.1.

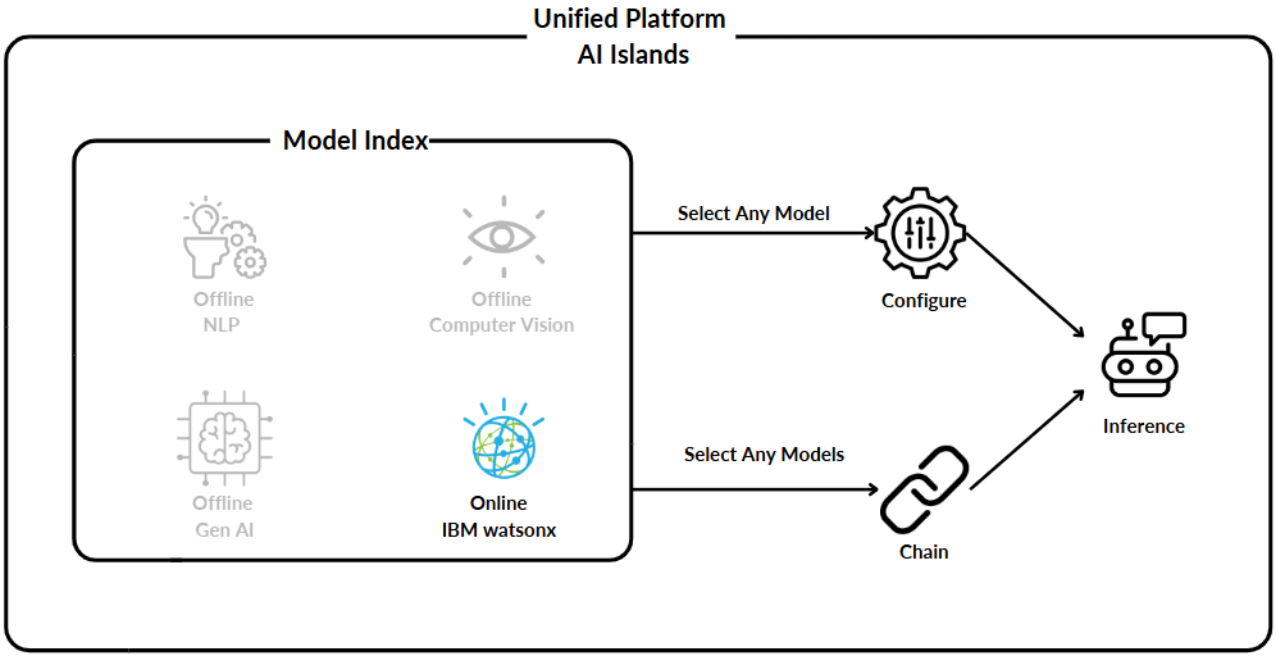


Figure 3.1: Visual Representation

In practice, we want the ability to select any model of any type from an AI model index and perform main model interaction features that are common to all models. Additionally, we want the ability to select any group of any models from the index, chain them together, and inference the model chain.

3.4 Use Case Diagram

The diagram in Figure 3.2 shows the general use cases for AI Islands, with more respect to the IBM Cloud watsonx models, while some use cases not particularly relevant to watsonx have been abstracted away for transparency. A colour-coding grouping system has been used to collect use cases together for further clarity. Unified Modelling Language (UML) has been used to show stereotypes such as *includes* and *extends*, where certain use cases have dependencies on others.

There is only one type of AI Island user account, with two different actors; the user and the IBM Cloud (an external actor).

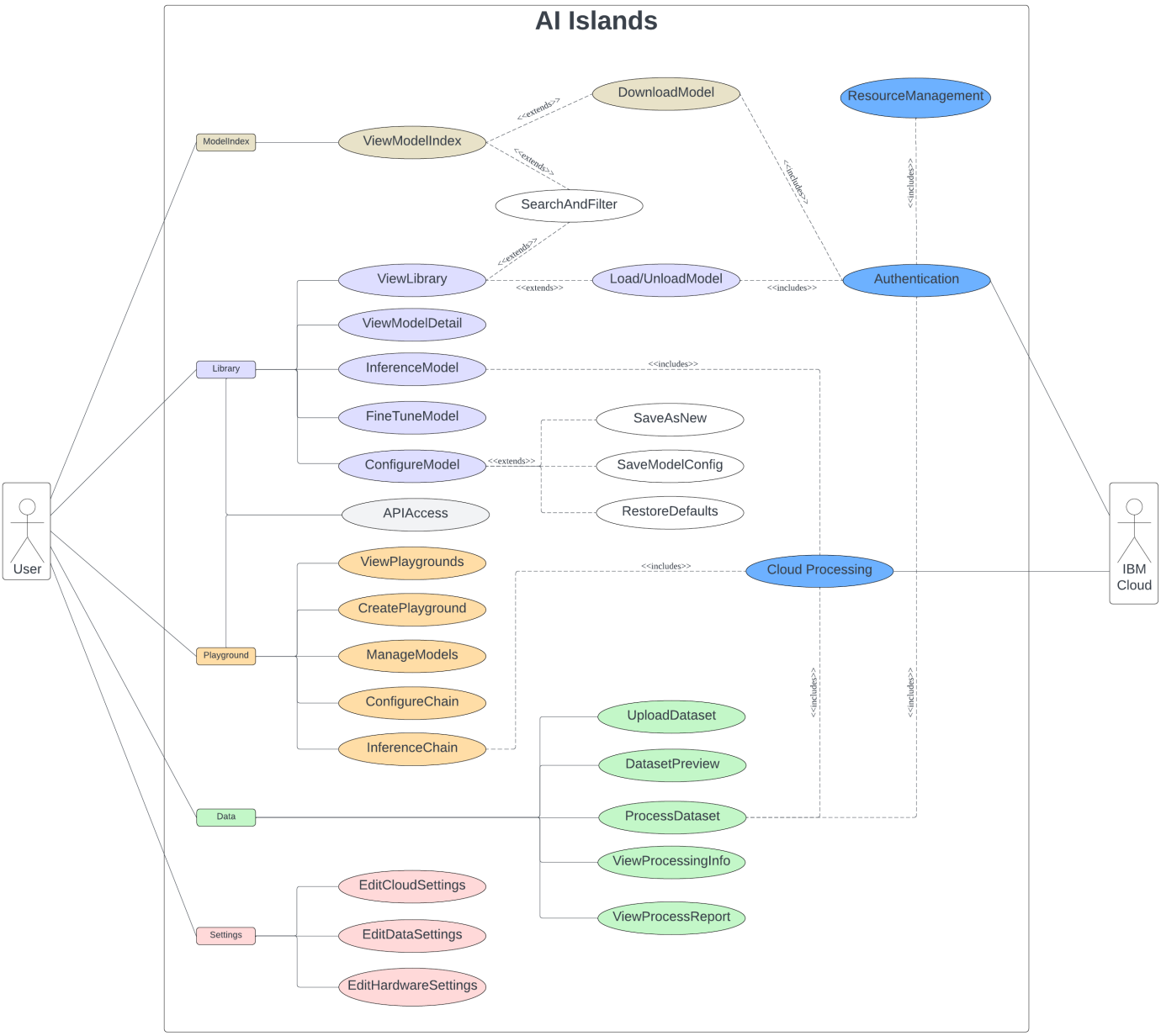


Figure 3.2: Use case diagram

3.5 Use Cases

Table 3.1 defines the ID for each use case presented in the use case diagram from Figure 3.2. The level of project relevance is shown on a scale of 1-5 in Table 3.1, where 5 is defined to be a use case exclusive to this particular project, 4 is defined to be a use case common to other sub-projects, such as 'DownloadModel', 3 is defined to be a supporting use case, 2 is defined to be a use case that has been tested, and 1 is defined to be a use case that was researched and experimented with.

For a full set of use case specifications, please see **Appendix B**.

ID	Use Case	Level	Actor
UC1	ViewModelIndex	4	User
UC2	DownloadModel	4	User
UC3	SearchAndFilter	4	User
UC4	ViewLibrary	4	User
UC5	Load/UnloadModel	4	User
UC6	ViewModelDetail	4	User
UC7	InferenceModel	4	User
UC8	FineTuneModel	1	User
UC9	ConfigureModel	4	User
UC10	SaveAsNew	4	User
UC11	SaveModelConfig	4	User
UC12	RestoreDefaults	4	User
UC13	APIAccess	2	User
UC14	ViewPlaygrounds	2	User
UC15	CreatePlayground	2	User
UC16	ManageModels	2	User
UC17	ConfigureChain	2	User
UC18	InferenceChain	4	User
UC19	UploadDataset	5	User
UC20	DatasetPreview	5	User
UC21	ProcessDataset	5	User
UC22	ViewProcessingInfo	5	User
UC23	ViewProcessReport	5	User
UC24	EditCloudSettings	5	User
UC25	EditDataSettings	5	User
UC26	EditHardwareSettings	4	User
UC27	Authentication	5	IBM Cloud
UC28	ResourceManagement	5	IBM Cloud
UC29	CloudProcessing	5	IBM Cloud

Table 3.1: Table of Use Cases

Project Relevance Levels ::= [5 = Exclusive, 4 = Shared, 3 = Supporting, 2 = Testing, 1 = Research]

3.6 MoSCoW Requirement List

The following sections, 3.6.1 & 3.6.2, outline the functional and non-functional requirements for the AI Islands application using the MoSCoW prioritisation method. This approach classifies the requirements into four categories: **Must have**, **Should have**, **Could have**, and **Won't have**, ensuring clarity on what features are essential over those that are optional or deprioritised.

Together, these requirements offer an overview of the system's expected functionality.

3.6.1 Functional Requirements

The Functional Requirements (FR) listed below in Table 3.2 define the capabilities and features that the AI Islands application provides, with varying priority. These requirements focus on core functionalities, such as model management, inference, configuration, and cloud integration, ensuring that users can effectively interact with AI models and perform tasks within the platform.

ID	Requirement	Priority
FR-1	The app shall feature a searchable and filterable AI model index consisting of both offline run models and watsonx cloud-based run models, spanning over computer vision, natural language processing, and text-generation AI model types.	Must have
FR-2	The app shall allow a user to download a model from the index, thereby adding the model to their library.	Must have
FR-3	The app shall feature a searchable and filterable model library consisting of all downloaded models.	Must have
FR-4	The app shall allow a user to load and unload a model.	Must have
FR-5	The app shall allow the user to inference a model.	Must have
FR-6	The app shall show the user their playground index.	Must have
FR-7	The app shall allow the user to load and inference a playground model chain.	Must have
FR-8	The app shall use cloud-based processing to run watsonx models on the user's IBM cloud account.	Must have
FR-9	The app shall allow the user to edit their IBM Cloud account settings.	Should have
FR-10	The app shall use account authentication and resource management to manage watsonx models.	Should have
FR-11	The app shall allow creation and deletion of playgrounds.	Should have
FR-12	The app shall allow the user to manage models and configure the model chain order within a given playground.	Should have
FR-13	The app shall allow the user to upload and preview a dataset.	Should have
FR-14	The app shall allow a user to process a dataset with embedding models to create files necessary for the RAG system.	Should have
FR-15	The app shall allow a user to view model details.	Should have
FR-16	The app shall allow a user to fine-tune a selection of models.	Could have
FR-17	The app shall allow a user to configure a model. The app shall allow the user to save this configuration as either a new model under a new model ID, or the current model. The user should be able to restore configurations from default if necessary.	Could have

FR-18	The app shall show API access information so that users can use third party applications to run models and model chains.	Could have
FR-19	The app shall show processing information within the app, and generate processing reports for further detail.	Could have
FR-20	The app shall allow the user to edit the RAG settings.	Could have
FR-21	The app shall allow the user to edit hardware settings.	Could have
FR-22	The app shall allow users to fine-tune watsonx models on their cloud account programmatically.	Won't have

Table 3.2: Table of Functional Requirements

3.6.2 Non-Functional Requirements

The Non-Functional Requirements (NFR) listed below in Table 3.3, set the quality standards and constraints for the AI Islands application. These include performance, scalability, usability, and security requirements, ensuring that the platform remains responsive, maintainable, and secure while offering a consistent user experience across use cases.

ID	Requirement	Priority
NFR-1	The app shall allow easy updates to the model index.	Must have
NFR-2	The app shall provide a responsive and user-friendly interface, optimised for different screen resolutions.	Must have
NFR-3	The app shall be scalable to accommodate the addition of new AI models from new model sources, e.g. OpenAI.	Should have
NFR-4	The app code base shall be maintainable, with new feature expandability.	Should have
NFR-5	Support for all major operating systems (Windows, macOS, Linux) to ensure broad accessibility.	Should have
NFR-6	The app shall maintain consistent style, layout, and user experience across all sections e.g. Library, Playground.	Should have
NFR-7	Layout and functionalities shall prioritise ease of use, with intuitive navigation and clear instructions for configuring and running models.	Should have
NFR-8	The app shall ensure data security and privacy, particularly when handling API keys e.g. masking.	Should have
NFR-9	The app shall include error handling and logging to provide detailed information for debugging and user support.	Should have
NFR-10	The app shall process user inputs and requests efficiently, with feedback provided for processes that take longer than 10 seconds (e.g., model downloads).	Could have
NFR-11	The app shall efficiently manage system resources, allowing for the smooth operation of both CPU and GPU-intensive tasks.	Could have

Table 3.3: Table of Non-Functional Requirements

3.7 Chapter Summary

In this chapter, we introduced the stakeholders and the methods used to gather the requirements for the AI Islands project. Through stakeholder meetings, IBM technical consultations, and document analysis, we identified the necessary features and goals for the project. The project brief was discussed, leading to iterative changes that shaped the project's scope, particularly in integrating watsonx services into AI Islands.

We then presented a use case diagram and a detailed list of the use cases, each with defined project relevance levels. These use cases guide the functionality of AI Islands, focusing on managing AI models, performing inferences, and integrating IBM services.

Finally, we outlined the MoSCoW Requirement List, including both Functional and Non-Functional Requirements. The functional requirements highlight essential capabilities like model management, inference, configuration, and cloud integration. The non-functional requirements address quality attributes such as scalability, usability, and performance, ensuring that the AI Islands application remains maintainable and responsive.

We can reinforce the connection between the use cases and requirements using a *use case traceability matrix*. Please see Table B.31 in Appendix B.

Chapter 4

HCI Design

In this chapter, we will analyse specific user use cases through a persona and scenario analysis, while linking these to AI Islands. We will then look into the initial design phase for the application, finishing with a simplified navigational map to better explain the application structure.

4.1 Persona and Scenario

In this section, we introduce three personas, each representing a different type of user for the AI Islands platform. Personas are fictional characters created to model the behaviors, goals, and challenges of real users, helping to illustrate how different types of individuals interact with the system. By defining these personas, we can better understand the needs of varied users and tailor the platform's features and UI accordingly.

Each persona is paired with a scenario, explaining how the user would engage with AI Islands. These scenarios provide real-world examples of how users with different goals and hardware access levels can benefit from the platform's diverse AI model index. By mapping these scenarios to specific AI tools, we can demonstrate the practical application of the system for solving user-specific tasks.

The persona cards in **Figures 4.1, 4.2, and 4.3** all contain background information necessary to understand the scenario. Computer hardware access is displayed as this is a factor determining if offline or online models should be used. If the person has a low hardware access score, they will tend to use online models, and if they have a high score they have the ability to use offline models. Additionally, cost is a factor, in that a person looking for free AI tools will tend to use offline models.

All personas in this section assume the use of the standard AI Islands user account, which is the only account type available. Additionally, for scenarios involving cloud-based models, it is assumed that the personas have already set up their IBM Cloud settings with the required account resource management; a topic that will be covered in later chapters.

4.1.1 Case 1: Business Employee

Persona: A business employee often handles many datasets within their business. They will need AI dataset tools to analyse various documentation within a fast-paced working environment where answers are required quickly. This type of user would require the RAG system functionality which would be able to provide accurate query results based on dataset information. Depending on the hardware availability, the business employee can opt for either online or offline models to run their RAG queries with.

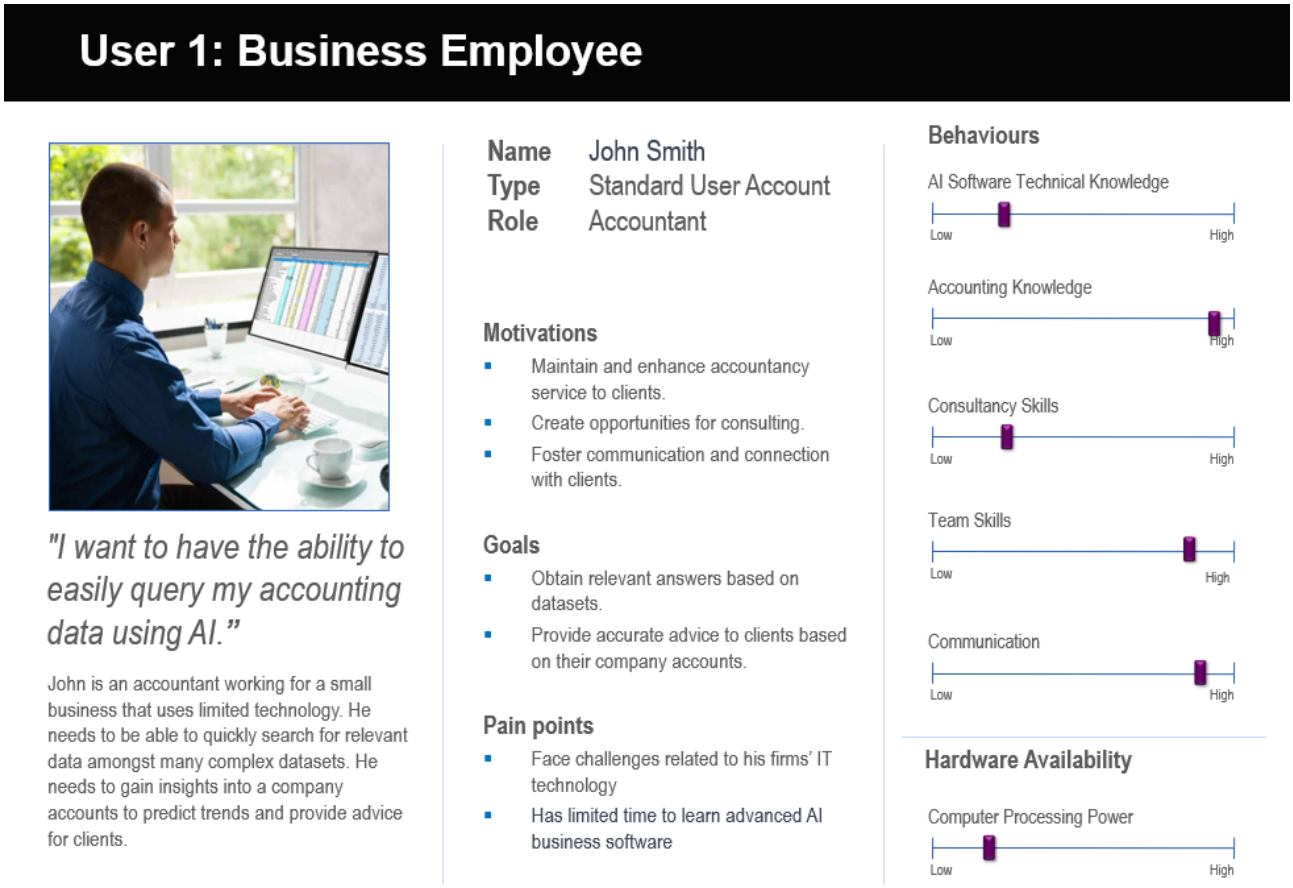


Figure 4.1: Persona: Business Employee

Scenario: John Smith has many client accounts stored in CSV and PDF format, along with relevant accounting regulation documents. He wishes to produce a summary of the expenses section within the accounting document. Due to his poor computer hardware, he downloads IBM's Granite; an online model. He proceeds to navigate to the dataset management area in AI Islands, uploads the accounts file, and then processes it using a watsonx embedding model. He then changes the configuration of Granite to accept this dataset as context, and to turn on the chat-bot feature. He then queries Granite to produce a full summary of the expenses.

John then decides to ask Granite how the company expenses could be reduced. Granite responds with various methods based on the context originally given (e.g. transport costs are too high), while providing comparisons to its knowledge base for similar business, providing insights into potential consultancy points.

4.1.2 Case 2: Student

Persona: A student often requires AI tools to assist in their studies. This type of user could require natural language processing (NLP) tools and text generation (LLMs) for report writing. Advanced passages of text can sometimes be difficult to grasp, especially due to a language barrier, highlighting the need for language analysis tools to provide accurate insights.

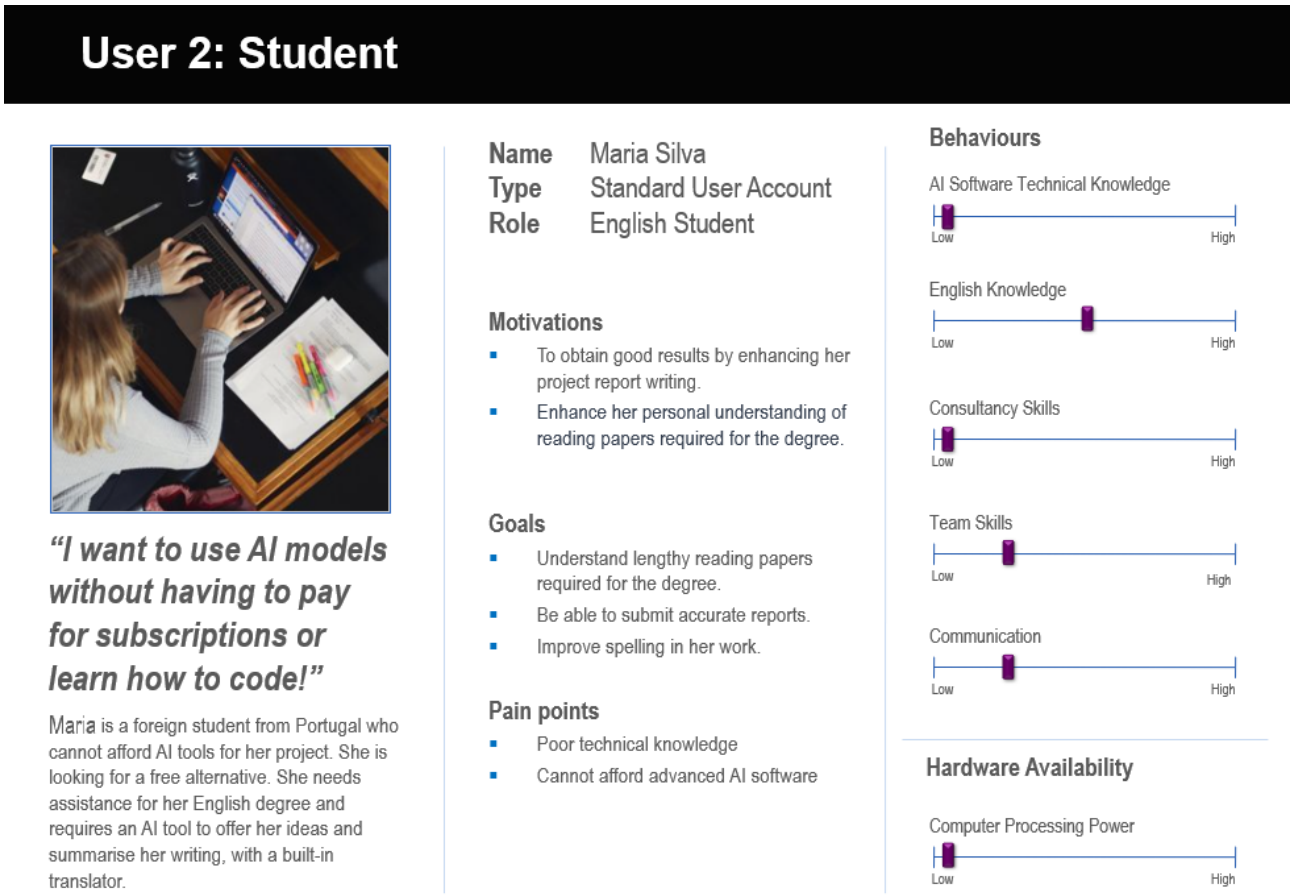


Figure 4.2: Persona: Student

Scenario: Maria is tasked with analysing two soliloquies written by Shakespeare and producing a short essay on their similarities and differences. She downloads IBM's Natural Language Understanding (NLU) and IBM's Granite, and loads them both. She then prompts NLU to analyse both soliloquies, and the model returns an analysis for each. She then prompts Granite with these two analyses to write an analysis paper on similarities and differences between Shakespeare soliloquies, highlighting the main concepts between both passages.

4.1.3 Case 3: AI Researcher

Persona: An AI researcher often requires access to high-powered technology in order to experiment with new research ideas. In more recent times, there has been a shift in focus to offline AI tools, in particular well-trained LLMs. However, researchers are interested in optimisation through configuring, prompting and chaining models together to better improve their performance.

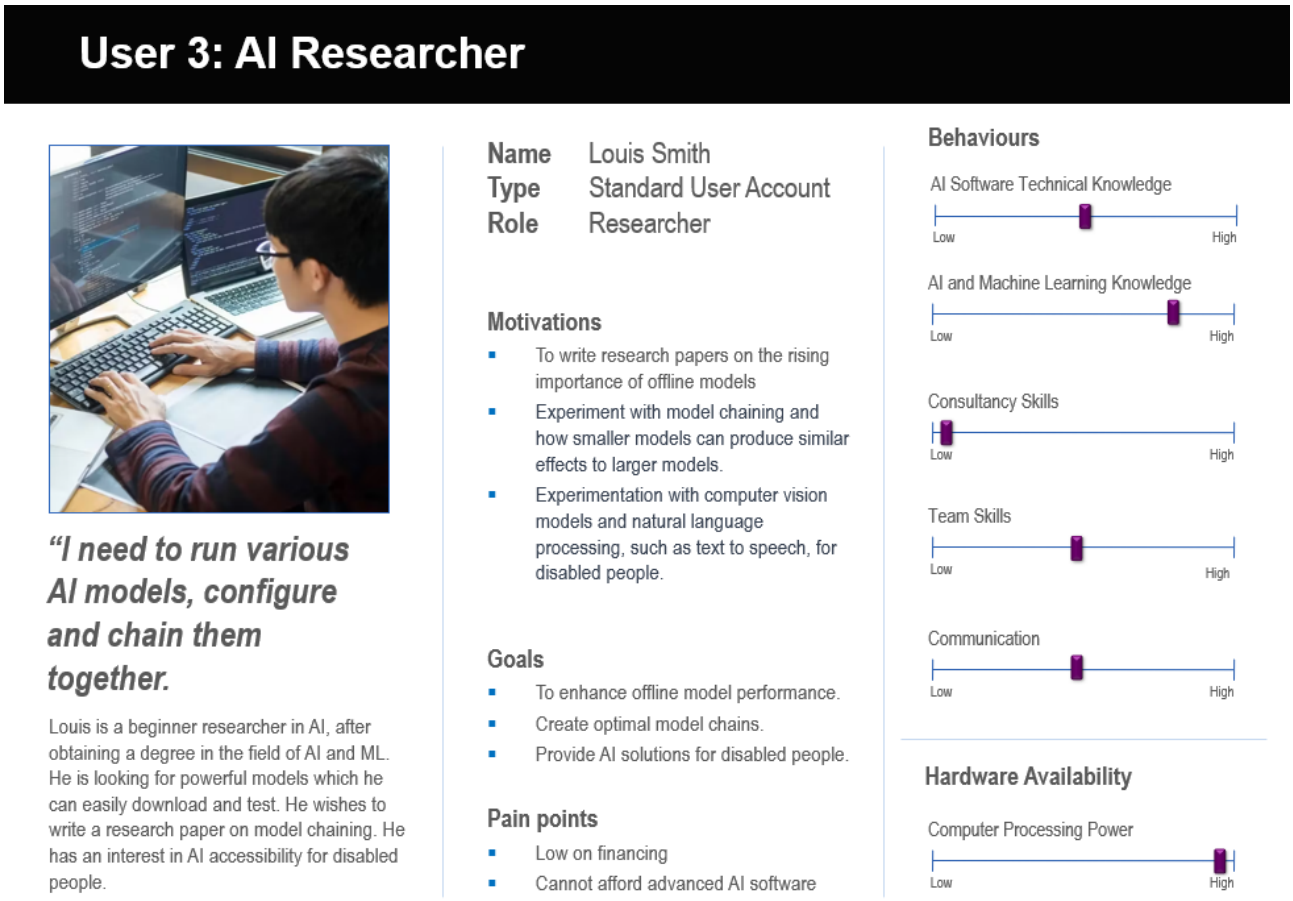


Figure 4.3: Persona: AI Researcher

Scenario: Louis is tasked with creating an offline AI tool that can generate an analysis of medical scans for patients, as part of an industry exchange network project. Due to the sensitive nature of this data, his client has asked for the tool to be offline. Louis searches the AI Islands model index to find a computer vision (CV) model and an offline LLM. He downloads both and creates a playground titled 'medical scans tool' in which he orders the models so that the CV model is first, and the LLM is second. He tests the playground inference first by uploading a medical scan image, but finds the output inaccurate.

Louis customises each model to achieve more optimal results. He decides that the CV model should be fine-tuned on the clients' dataset of medical scans, which he does. He then changes the configuration and parameters of the LLM until he is achieving an optimised AI tool within the 'medical scans tool' playground. He then writes a paper, perhaps using the LLM due to his low consultancy skills, explaining his findings and sends this to his client.

4.2 UI Design

This section covers the UI design process, including the acknowledgement of the design principles in 4.2.1 and the Figma design prototype in 4.2.2, relating the design to use cases from Figure 3.2.

4.2.1 Design Principles

Applying design principles helps create designs that are not only visually appealing but also effective in conveying messages or solving problems for the intended audience.

Consistency: A uniform design ensures a cohesive user experience, making it easier for users to navigate and interact with the application. Consistent placement of elements like navigation bars and standardised terms help avoid confusion.

Visibility: Key features and information should be easy to find. Using colors for keywords and highlighting navigation buttons helps draw attention to important elements. Readability is enhanced through appropriate font sizes and contrast.

Feedback: Providing feedback is essential, such as error pop-ups explaining why a process cannot be executed. Users are supplied with terminal pop-up windows where appropriate i.e. for downloads or fine-tuning.

Efficiency: The app is focused on its core purpose without unnecessary content or distractions. Every element is functional, helping users complete tasks without being overwhelmed by unnecessary features.

Learnability: The design is intuitive, following common conventions to minimise the learning curve. Navigation bars are clearly placed, and users can understand how to use the app without needing extensive guidance. Clarifications are provided in '?' buttons to support users where necessary.

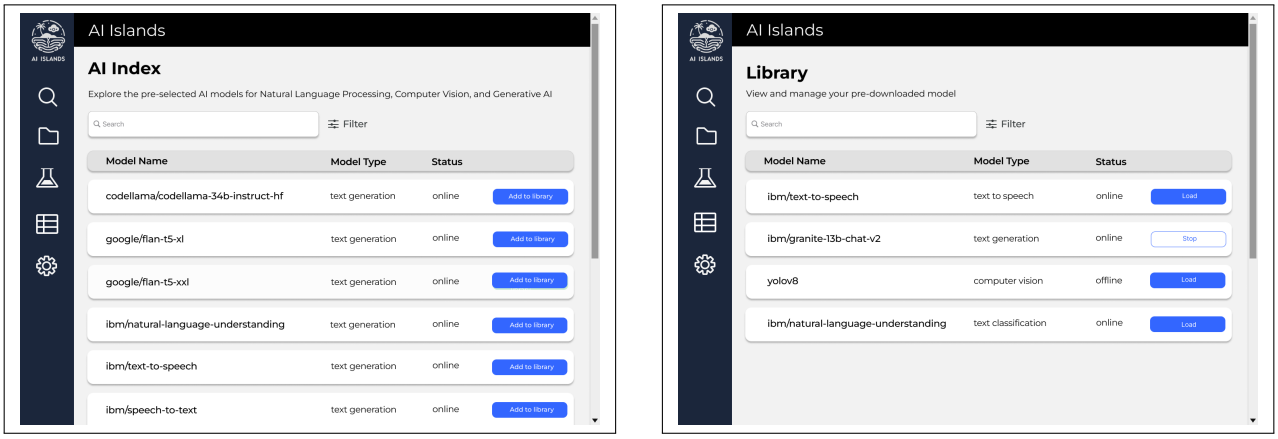
4.2.2 Application Mock-Up: Figma

Presented in a desktop application format, the mock-up figures aim to showcase the application design, layout, and functionality. Given users may have lower levels of technological expertise, the UI is kept simple and usable to deliver an optimal experience in accordance with Nielsen's '10 Usability heuristics for interaction design' [21]. The following use cases for the application's core features have been included in the mock-ups:

- UC1: ViewModelIndex
- UC4: ViewLibrary
- UC7: InferenceModel
- UC9: ConfigureModel
- UC14: ViewPlaygrounds
- UC17: ConfigureChain
- UC21: ProcessDataset
- UC24: EditCloudSettings

4.2.2.1 Figma Design: UC1 & UC4

A user opens the application and starts at the *AI Index*, see **Figure 4.4a**. Here, they can apply filters and search for a model by name or type. The user can download desired models by clicking 'Add to Library'. A user can then navigate to the *Library*, see **Figure 4.4b**. Here, they can view all of their downloaded models and load / unload them into memory.



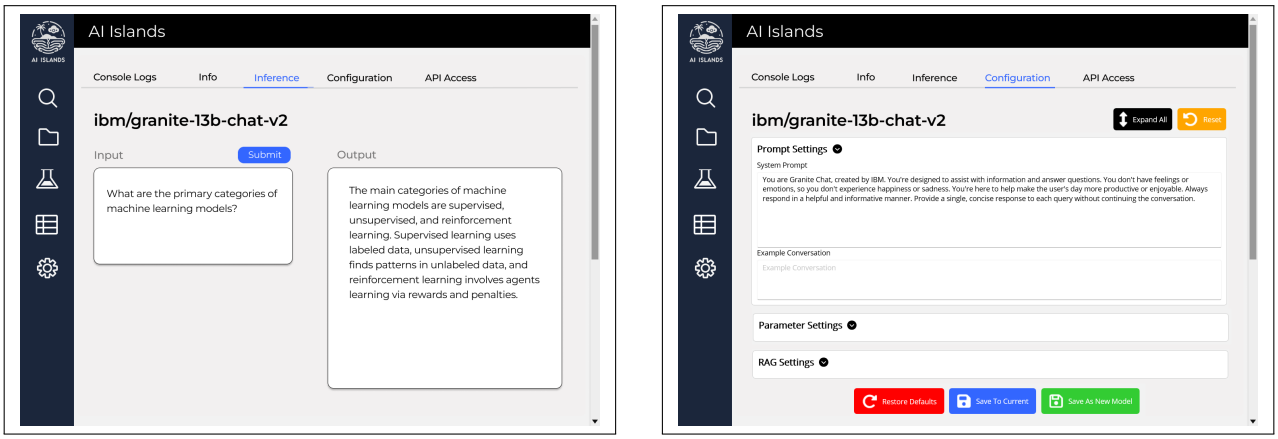
(a) UC1: AI Islands Model Index

(b) UC4: AI Islands Library

Figure 4.4: Browsing Pages

4.2.2.2 Figma Design: UC7 & UC9

A user clicks on a model within their *Library*, in this case IBM's Granite. They make their way to the *Inference* page, see **Figure 4.5a**. Here, they are able to use the model by submitting prompts. A user can navigate to *Configuration* where they can change the models' configuration, see **Figure 4.5b**.



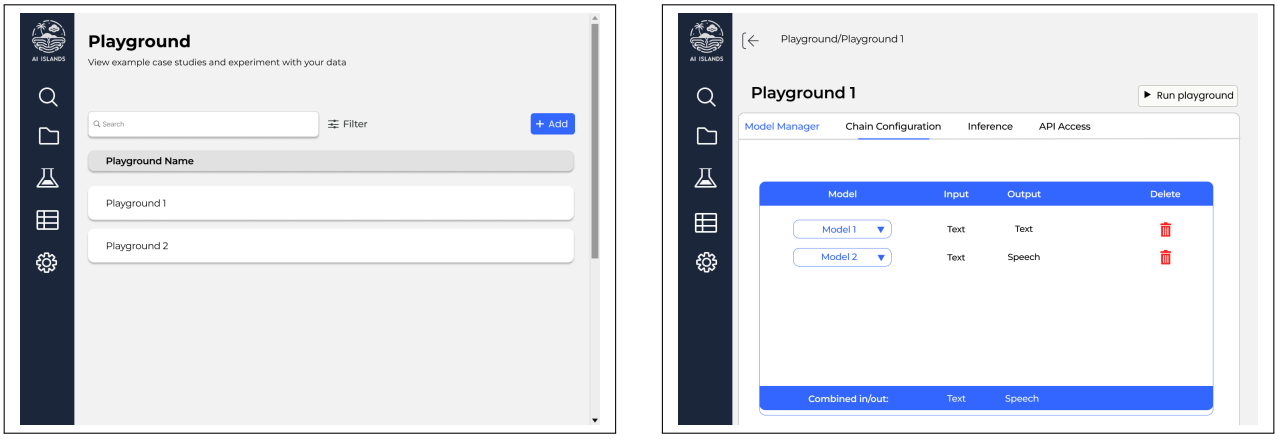
(a) UC7: AI Islands Model Inference

(b) UC9: AI Islands Model Configuration

Figure 4.5: Library Individual Model Feature Pages

4.2.2.3 Figma Design: UC14 & UC17

A user navigates to the *Playground* via the navigation bar. Here, they can view their playground list, see **Figure 4.6a**. They can click on a playground instance which takes them to a tabbed page view where they can access the *Chain Order* and edit, see **Figure 4.6b**.



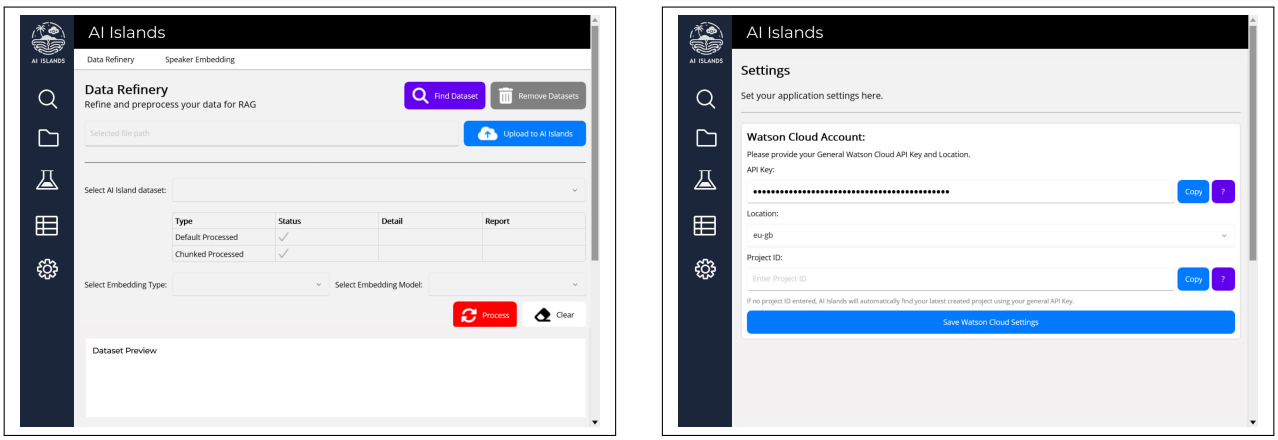
(a) UC14: AI Islands Playground List

(b) UC17: AI Islands Playground Chain Configuration

Figure 4.6: Playground Pages

4.2.2.4 Figma Design: UC21 & UC24

A user can optionally use the built in RAG system for context-based prompting within a text-generation model. They can navigate to the *Data Refinery*, in which they can upload datasets, preview them, and process them with locally run or cloud-based run watsonx embedder models, see **Figure 4.7a**. The user can also change their global settings by navigating to *Settings* in which they can edit their 'API key', 'Location', and 'Project ID'; all necessary to run IBM services within AI Islands, see **Figure 4.7b**.



(a) UC14: AI Islands Data Refinery

(b) UC17: AI Islands Settings Page

Figure 4.7: Auxiliary Pages

4.3 Navigation Map

There is only one navigation map for AI Islands since there is a single account type. On launching AI Islands, the first page to appear is the *AI Index* (i.e. model index) and the app shell consists of a navigation bar with the five main pages denoted by solid arrows. Once the user navigates to one of these pages, they obtain access to the sub-pages denoted by the dotted lines, see **Figure 4.8** below.

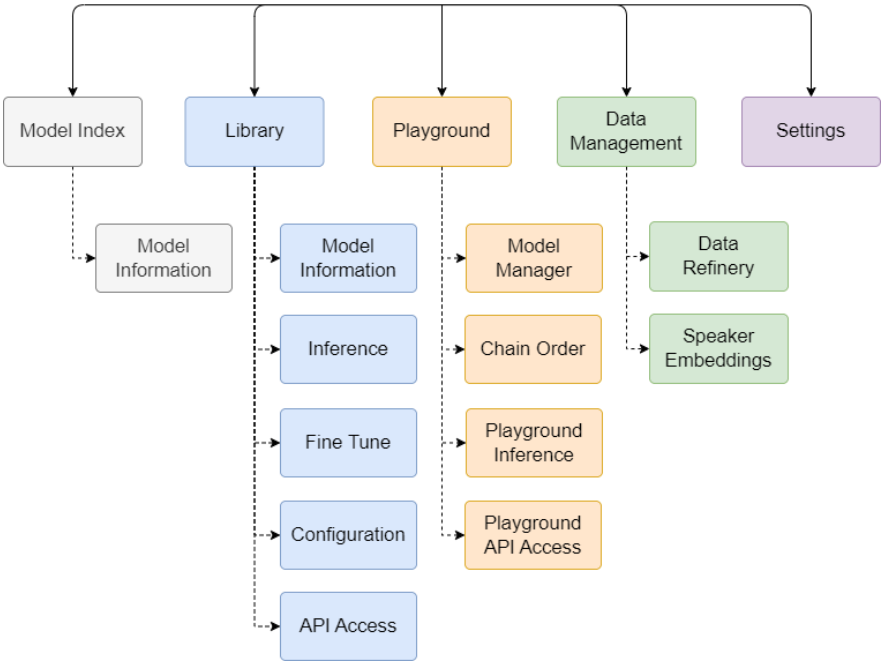


Figure 4.8: AI Islands Navigation Map

4.4 Chapter Summary

In this chapter, we analysed user interactions through a persona and scenario analysis, providing insights into how different types of users would benefit from AI Islands. These personas helped highlight the platform's flexibility in catering to varied hardware access levels and AI tool requirements, providing real-world scenarios for each user.

We then detailed the initial UI design phase, emphasising the application of core design principles such as consistency, visibility, feedback, efficiency, and learnability. The design mock-ups, created in Figma, demonstrated how core use cases (e.g., viewing the model index and inferencing models) are implemented in the platform's UI.

Finally, we presented a simplified navigation map to illustrate the application's structure, showing how users can access key features through a straightforward and intuitive interface.

Chapter 5

System Architecture

In this chapter, we will analyse the system architecture of the AI Islands project, focusing on backend and frontend technologies as well as database design. We will cover the backend object-oriented class structure, and FastAPI for frontend communication. We will then move to the frontend, highlighting the use of C# with .NET MAUI and the MVVM pattern. Lastly, we explore the use of JSON for storing application data in a semi-structured database.

5.1 Backend Design

This section will cover the backend design choices for AI Islands. We will cover the technology used and the structure adopted for main backend functionality.

5.1.1 Python

Python was chosen as the backend language for AI Islands due to its support for machine learning and AI libraries. It offers a wide range of tools and frameworks that simplify AI model development and integration [22]. Additionally, all required IBM libraries, essential for incorporating watsonx services programmatically, are available in Python, and advertised as such on the IBM documentation [23], further solidifying it as the optimal choice in order to incorporate watsonx.

An alternative considered was Java, which also supports some AI libraries. However, Java has fewer libraries with less flexibility, particularly for fine-tuning models. Python allows us to easily expand features in AI Islands for future development, making it the better choice [24]. Additionally, many IBM services do not have the same level of support in Java, making Python the clear choice.

5.1.2 Class Structure

Due to the project's focus on integrating diverse AI models into a unified system, it was important to create a structured and scalable backend design. We opted for a control class structure whereby the *control* class would generalise a process, and its *child* class (following the *base model*) would provide the functionality relevant for a specific model type.

This can be easily seen with the **ModelControl** class which provides control methods for the model type classes, namely TransformerModel, UltralyticsModel, WatsonModel, and WatsonService, see **Figure 5.1**. This design allows for future expandability; if a developer wishes to add a new type of model that requires different methods for downloading, loading, and inferencing, they would simply create a new model class that follows the *base model*.

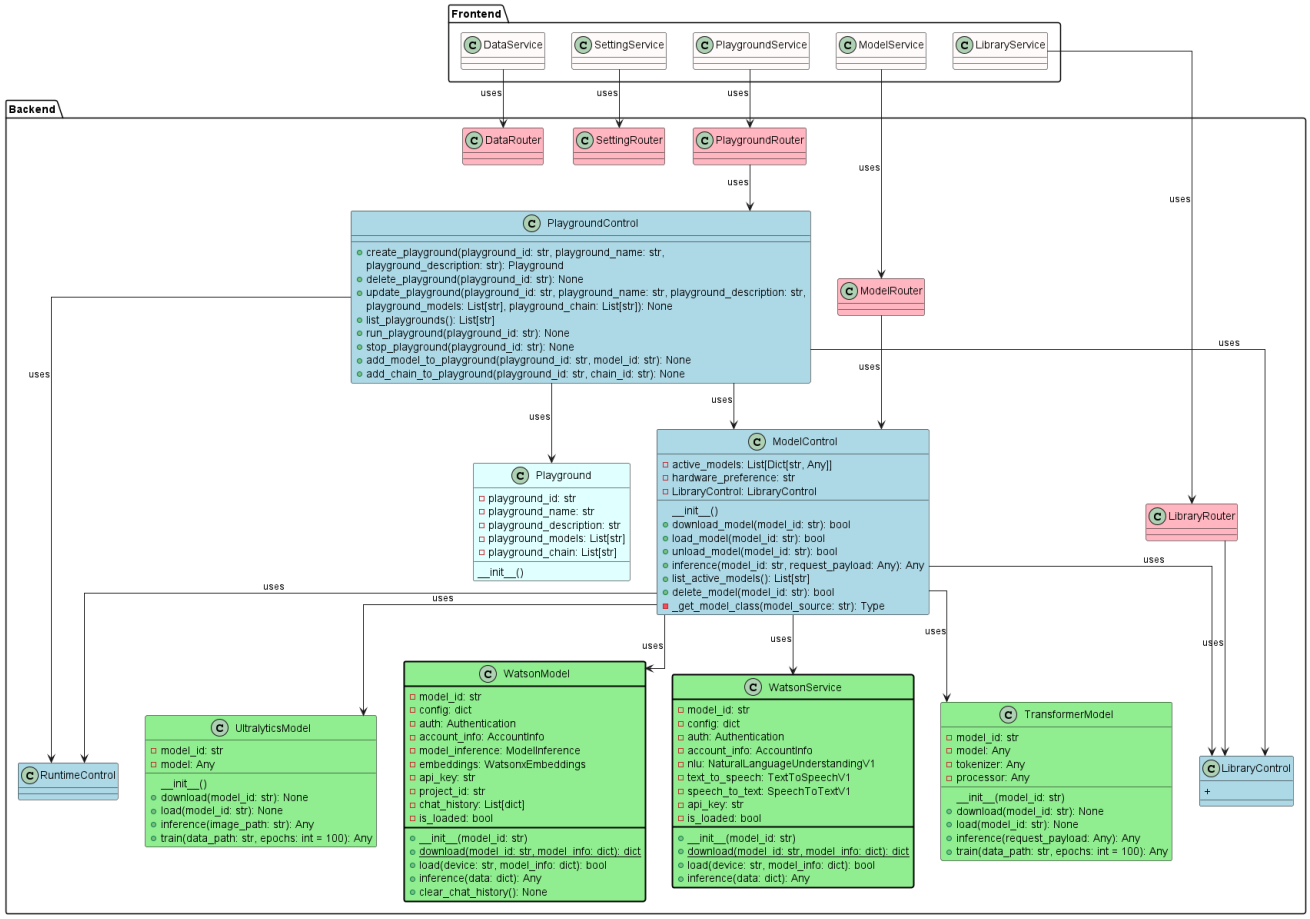


Figure 5.1: Backend Object Oriented Design Diagram

In this report, we will be focusing on the **WatsonModel** and **WatsonService** classes, and a more detailed analysis of these classes will be provided in Chapter 7, Sections 7.1.1.1 & 7.1.1.5.

5.1.3 FastAPI

FastAPI was chosen as a technology required to communicate with a frontend. It allows for easy 'swapping out' of the frontend to allow for a different technology to be used, and in our case, allows a single application to be built using a mix of Python and C#. The FastAPI is used on the 'surface level' of the backend, as seen in Figure 5.1, denoted by the pink router classes.

These routers, defined in separate modules, encapsulate related API endpoints for different functionalities. Each router module typically contains a Router class that defines specific routes and their corresponding handler methods. While some routers, such as ModelRouter, have associated control classes (i.e. ModelControl) that implement core business logic, DataRouter is associated with utility classes for dataset processing and file management, and SettingsRouter with settings related utility classes.

This well-organised structure promotes easy expansion of functionality. New features can be added by creating additional route modules, with or without corresponding control classes.

5.2 Frontend Design

This section will cover the frontend design choices for AI Islands. We will cover the technology used and the structure adopted for main frontend functionality.

5.2.1 C# & .NET MAUI App

Due to stakeholder requirements, it was instructed to create a frontend in C#. Research was conducted into other options, such as python-based Flask, Flet, and Tkinter. A strong alternative was to use Electron [25], but this was overruled by a stakeholder, due to its application memory usage.

MAUI (Multi-platform App UI) is a cross-platform framework that allows developers to build native applications for iOS, Android, Windows, and macOS using a single codebase. It follows the MVVM (Model-View-ViewModel) architectural pattern, which separates the logic into three key components:

- Models: representing the data.
- Views: xaml pages defining the UI.
- ViewModels: for logic and data binding between Models and Views.

We can see this structure in **Figure 5.2**. The 'App.xaml.cs' file acts as the app entry point, while the 'AppShell.xaml.cs' manages navigation to pages in 'Views'. We can then use logic files in 'ViewModels' which interact with the 'Services'. The service classes then communicate with the backend router classes. This structure allows for a clean separation of concerns.

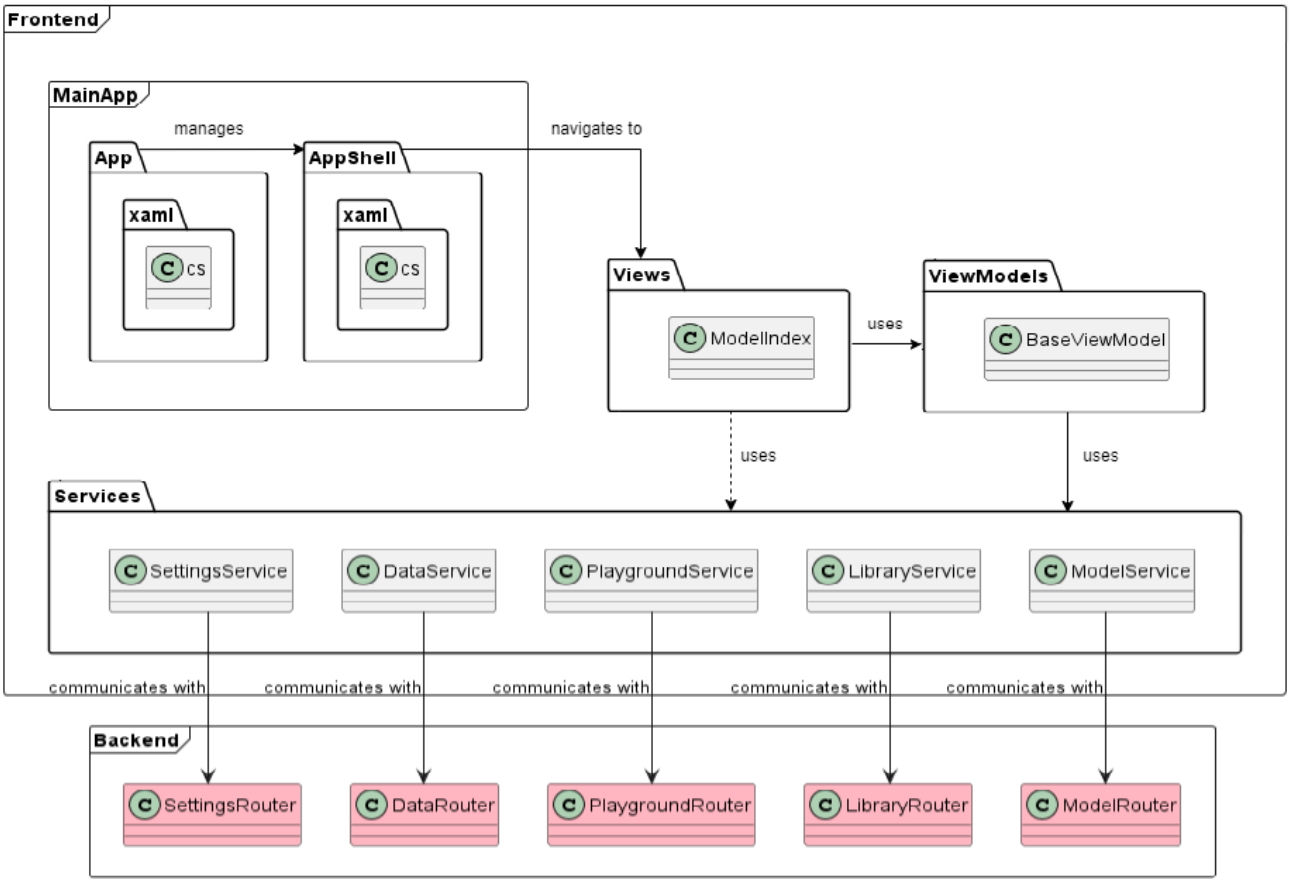


Figure 5.2: Front-end Structure Diagram

5.2.2 Service API

Due to the nature of using FastAPI within the backend, it made sense to mirror the router classes in the frontend, which avoids directly accessing end-points within the 'ViewModel' logic files. Instead, we can define C# functions to handle these end-points. You can see from the two diagrams in Figure 5.2 and Figure 5.1 that the service classes are handling the end-points in their respective router file, i.e. 'ModelService' in the front-end, is communicating with 'ModelRouter' in the backend.

5.3 Database Design

As mentioned in earlier chapters, AI Islands is a collection of completely different AI models working together within a unified platform. When it comes to the deeper levels of the application, where we need to be able to store model configurations for different model types, we need to adopt a flexible database method.

5.3.1 Semi-structured JSON

JSON (JavaScript Object Notation) files are simple, flexible, and widely supported, making it an ideal format for managing and configuring AI models, allowing for easy data manipulation, storage, and exchange between different components of AI Islands. Here are some reasons why they are used in this application:

1. **Structured:** JSON provides a clear, nested structure for representing complex data. This is particularly useful for storing model configurations.
2. **Readable:** JSON is easy for developers to read and write, while being simple for machines to parse and generate.
3. **Language-agnostic:** JSON is supported by most programming languages, making it easy to share data between different parts of the application, see subsection 5.3.2 .
4. **Flexible:** JSON allows for nested structures and arrays, which is useful for representing complex model configurations, such as model parameters and pipeline settings.
5. **Updatable:** JSON files can be easily updated programmatically, which is useful for managing a dynamic collection of AI models
6. **Interoperable:** JSON can be used to exchange data between different services and APIs, which is crucial when working with various AI models and IBM Watson services.

5.3.2 JSON In C# Typing

As discussed earlier in this chapter, the MAUI application uses an MVVM structure to separate out concerns. Due to the strongly typed nature of C#, we need to parse the JSON data correctly through a 'Model' type file. In this case, we can analyse 'Model.cs' within the 'Models' folder in the front-end. (Note that due to the nature of AI Islands handling AI models, this file has been named the same as the structural file type by coincidence.) In the file, it can be clearly seen that the data from the AI model is being handled and prepared for use in the frontend logic files.

5.4 Chapter Summary

In this chapter, we have analysed the technologies and code file structure, while justifying the choices made. It is clear that the design promotes expandability by the use of control classes; the developer would need to make a new Model Class to allow different models through AI Islands. We can combine all of these components together within a high-level architecture diagram with respect to basic model functionality, see Figure 5.3.

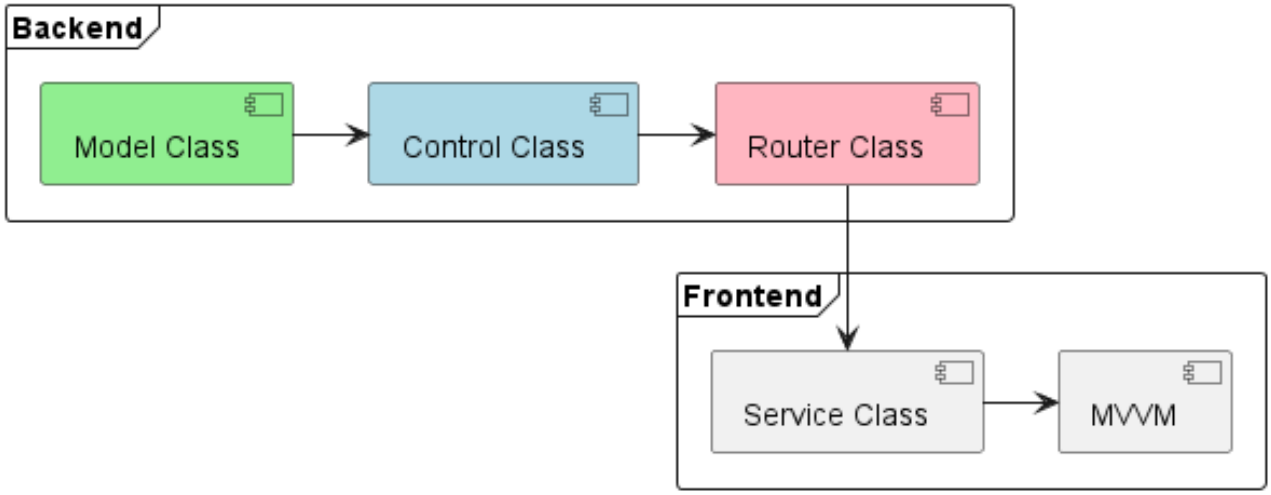


Figure 5.3: High-level Design Architecture

Chapter 6

Implementation I: Watsonx Programmatic Techniques

In this chapter, we will reference the IBM Cloud Guide from **Appendix C**.

For the technical reader interested in understanding the details more thoroughly, a collection of Python Jupiter notebooks are provided as supplementary material for the implementation chapters, titled **Integrating Watsonx: A Programmatic Guide**. Additionally, it is advised that the reader creates an IBM Cloud account to explore the information provided in these chapters.

6.1 IBM Cloud Authentication

In this section, we will analyse IBM Cloud authentication, gradually building towards solutions due to the complexity of the various authentication methods. Specifically, we will cover the challenges and solutions related to authentication for IBM watsonx AI and IBM services. Along the way, we will justify key decisions that led to more unified authentication methods.

6.1.1 Authentication Method - API Keys

We will begin by introducing API keys, their role, and how they function in coordination with the IBM Cloud. We will identify how vital they are for integration with watsonx.

An API key is a type of secret key that allows programmatic access to a specific user account, enabling the use of a company's services. In our case, IBM provides programmatic access through both API keys and service URLs.

IBM offer users to create general account API keys through their Cloud Identity and Access Management (IAM) feature. From the top navigation bar on their cloud account, they can navigate to 'Manage', 'Access (IAM)', 'API Keys', and can create a new key here, as seen in **Figure C.6** [26]. The reader should complete this step now, before attempting to run supplementary material.

This type of key is required to programmatically use watsonx foundation models and embedder models. However, for IBM services, such as Natural Language Understanding (NLU), it is more common to use an API and URL specific to the service instance [27], as seen in **Figure C.5**. This creates an issue when trying to uniformly integrate IBM services together within a programmatic desktop application.

6.1.2 Authentication Library Packages

As mentioned in 6.1.1, there is a difference between authenticating with watson machine learning based models and authenticating with packaged services.

6.1.2.1 Foundation and Embedding Models Authenticating Library

IBM supplies the **'IBM Watson AI'** python library for setting up LLMs and embedders programmatically. We use the **Credentials** class from IBM's Watson AI to connect to watsonx AI models, as seen in Listing 6.1.

```
# Import the authentication class
from ibm_watsonx_ai import Credentials

# Set the api key associated with the cloud account
IBM_CLOUD_API_KEY='your-api-key-here'
# Set the service url 'https://<location>.ml.cloud.ibm.com'
IBM_CLOUD_MODELS_URL='https://eu-gb.ml.cloud.ibm.com'

# Initialise the client
credentials = Credentials(IBM_CLOUD_MODELS_URL, IBM_CLOUD_API_KEY)
```

Listing 6.1: Watsonx.ai Authentication

We can then inference the model using the credentials client object, as seen later in 6.3.2.

6.1.2.2 Service Authenticating Library

IBM supplies the **'IBM Cloud SDK Core'** python library for setting up IBM services programmatically. We use the **IAMAuthenticator** class from IBM's Cloud SDK to connect to watson services, as seen in Listing 6.2.

```
# Import the authentication class
from ibm_cloud_sdk_core.authenticators import IAMAuthenticator

# Set the api key associated with service instance
IBM_SERVICE_API_KEY='your-service-specific-api-key-here'
IBM_SERVICE_URL='your-service-specific-url-here'

# Initialise the client
authenticator = IAMAuthenticator(IBM_SERVICE_API_KEY)
```

Listing 6.2: Watson Service Authentication

This is already presenting us with a problem. The watsonx AI models are using the user general API Key, but the watson services are using specific service instance API Keys to authenticate.

If we were to continue this method and set up the service, we would use the code in Listing 6.3.

```
# Import the watson service
from ibm_watson import WatsonService

# Set up the service using the authenticator object previously set
ws = WatsonService(version, authenticator)
# Use the in-built method to set the user specific service url
ws.set_service_url(IBM_SERVICE_URL)
```

Listing 6.3: Watson Service Authentication

Our problem has got worse! Of course we could use this direct approach, but it would require the AI Islands user to enter their specific API keys and service URLs for every single resource; this is not user-friendly.

We need to explore and identify a method that can obtain service-specific API keys and service URLs from a users' general IAM API Key. This would mean that all the AI Islands user would have to do is enter one single API key to access their resources programmatically.

6.1.3 IBMs IAM Token

IBM offers the ability to generate a user token based on an account general API key [28]. We can use the IBM identity token endpoint and define a function that returns a user token from their general API key alone, as seen in the following Listing 6.4.

```
# IBM Cloud IAM endpoint
IAM_TOKEN_URL = "https://iam.cloud.ibm.com/identity/token"

# Function to get IAM access token
def get_iam_token(key):
    headers = {
        'Content-Type': 'application/x-www-form-urlencoded',
        'Accept': 'application/json',
    }
    data = {
        'grant_type': 'urn:ibm:params:oauth:grant-type:apikey',
        'apikey': key,
    }
    response = requests.post(IAM_TOKEN_URL, headers=headers, data=data)
    token = response.json()['access_token']
    return token

# Call
IAM_token = get_iam_token(IBM_CLOUD_API_KEY)
```

Listing 6.4: IAM Token Generation

We can use this token for various operations, such as fetching account specific information and for validating API keys by simply attempting to fetch an IAM token. However, in this chapter we will focus on its use for allowing access to retrieve service specific API keys and URLs.

6.1.4 Resource Controlling with IAM Tokens

We can design a function to retrieve the service-specific credentials for a given IBM Cloud service using the IAM token, which then iterates through the response and finds the information required. Due to its length, we will show a pseudo-like code representation of the list iteration and dictionary access part to explain the concept, see Listing 6.5. The complete function is featured in the supplementary notebooks and AI Islands source code under the name *get_service_credentials()*.

```
# IBM Cloud IAM endpoint
RESOURCE_SERVICE_URL = "https://resource-controller.cloud.ibm.com/
                        v2/resource_instances"

# Function to retrieve service credentials using the IAM token
def get_service_credentials(iam_token, service_name):
    headers = {
        'Authorization': f'Bearer {iam_token}',
        'Accept': 'application/json',
    }
    response = requests.get(RESOURCE_SERVICE_URL, headers=headers)
    # After receiving the response:
    # - If the response is successful, extract resources from response.
    # - Search for the resource that matches the service name.
    # - If a match is found, retrieve the resource instance ID.
    # - Use the instance ID to request the resource keys (credentials).
    # - Return the first set of credentials if found.
```

Listing 6.5: Service Specific Credentials Function using the IAM Token

The function first queries the IBM Cloud Resource Controller API to get a list of all resource instances associated with the account. It then filters this list to find the instance matching the specified service name. Once the instance is found, it makes a second API call to retrieve the resource keys for that specific instance. The function returns the first set of credentials found for the service, which includes the service specific API key and service URL.

We can now return to the methods seen in Listings 6.2 & 6.3. Instead of using global variables consisting of service specific API keys and URLs, we can access the data from the credentials dictionary created from IAM token resource controlling in Listing 6.5, using dictionary key access. We can now set up a watson service using this kind of authentication.

```
# Obtain the full set of service specific credentials
credentials = get_service_credentials(IAM_token, ServiceName)

authenticator = IAMAuthenticator(credentials['apikey'])
ws = WatsonService(version, authenticator)
ws.set_service_url(credentials['url'])
```

Listing 6.6: Authenticating via IAM Token Resource Controlling

This approach allows for programmatic access to service-specific credentials without requiring tedious manual input, promoting easier integration of differing IBM Cloud services in applications.

6.1.5 Authentication Summary

In this section, we have looked at the problems associated with the direct authentication approach for watson services and how to overcome these. For AI Islands, the choice taken was to unify authentication, thus requiring the user to only enter one API key within AI Islands. The methods within this section form the basis of watsonx authentication within the AI Islands code. A sequence is provided in Figure 6.1 to illustrate the unified authentication technique adopted in AI Islands.

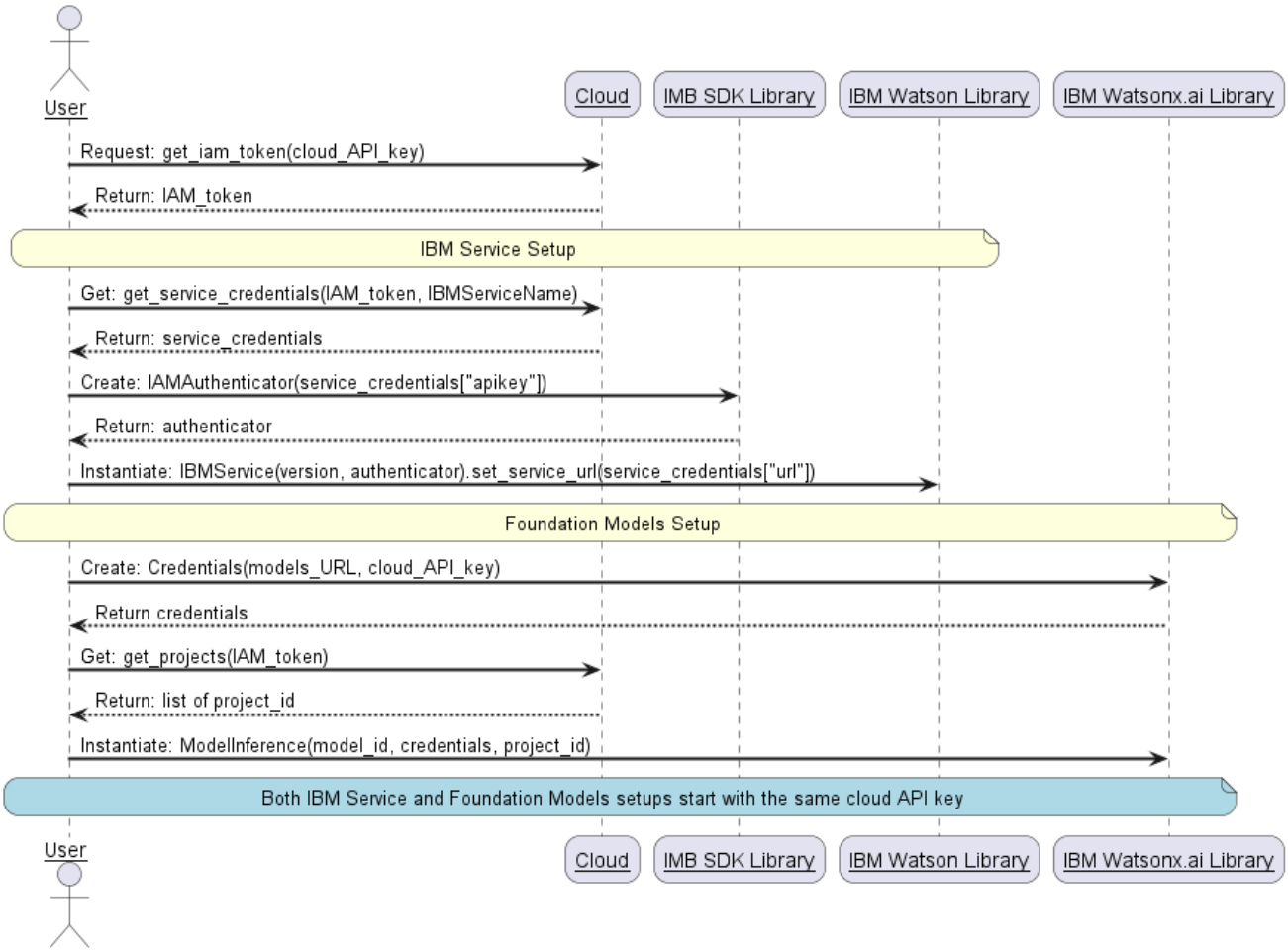


Figure 6.1: Unified Authentication Sequence Diagram

6.2 Programmatic Information Access

In this section, we will cover user account resource management. We will then explore both IBM Cloud user account information and service-specific data retrieval, highlighting how they can enhance the implementation of AI Islands.

6.2.1 IBM Command Line Interface (CLI)

The IBM Cloud Command Line Interface (CLI) is a powerful tool for interacting with IBM Cloud services directly from your terminal, you can see it in action within Appendix D. It allows retrieval of account resource information, without the need to go through the web interface.

Once logged in, we can use the basic commands seen in Listing 6.7. Note that fetching details in command Listing 6.7(2) is particularly useful for programmatically checking what resources have been added to a user account, whereas in command Listing 6.7(5) this is useful for checking the plan details (lite or standard paid).

```
# To list all resource instances in an account resource list
C:\Users>ibmcloud resource service-instances

# To fetch details about a particular service instance
C:\Users>ibmcloud resource service-instance "Watson Machine Learning-2m"

# To check account billing usage
C:\Users>ibmcloud billing account-usage

# To output a JSON containing resource information:
C:\Users>ibmcloud resource search "type:*" --output json
```

Listing 6.7: Basic CLI Commands

We need a similar method in python to be featured within the AI Islands application to implement useful validation, particularly for model set-up.

6.2.2 IBM Cloud Account Resource Management

We need to ensure we have added and configured all of the required resources for AI Islands within an associated IBM Cloud account in order to fully experience the online model index within the app.

6.2.2.1 Resource Existence

In order for online IBM models and services to work programmatically, the user must ensure they have created an instance of the required services on their account, see **Figure C.4**. This can be achieved by accessing the catalog. For access to all online models, the user must have the following resources added:

- For watsonx AI:
  - Watson Studio (WS)
  - Watson Machine Learning (WML)
  - Cloud Object Storage (COS)
- For watson services:
  - Natural Language Understanding (NLU)
  - Text to Speech (TTS)
  - Speech to Text (STT)

6.2.2.2 Resource Configurations

When it comes to watsonx AI models, the user will need to set up some initial configuration. We essentially need to replicate how the *prompt lab* is working within WS. Users will need to ensure they have an existing project within WS, and if not they must create one. They must additionally check that the WML service is integrated into the project space, which can be achieved as shown in **Figure C.10**.

6.2.3 Programmatic Access for IBM Account Information

As discussed earlier in this chapter, access to account resource listings and WS projects is vital for enhancing the AI Islands experience. Using IAM tokens with the resource controller endpoint allows us to retrieve an account's resource list, while the data platform projects endpoint enables us to fetch a user's project list; necessary for running watsonx AI models. The IAM token method, as demonstrated in Listing 6.4, can be applied for this purpose as shown below in Listing 6.8.

```
# IBM Cloud resource endpoint
IBM_CLOUD_PROJECTS_URL='https://api.eu-gb.dataplatform.cloud.ibm.com'

def list_projects(iam_token):
    headers = {
        "Authorization": f"Bearer {iam_token}",
        "Content-Type": "application/json"
    }
    endpoint = f"{IBM_CLOUD_PROJECTS_URL}/v2/projects"
    response = requests.get(endpoint, headers=headers)
    projects = response.json().get('resources', [])
    return projects
```

Listing 6.8: Account Information

The method seen in Listing 6.8 is a viable choice within AI Islands validation to prompt the user that they need to add a particular resource to their account if it does not exist.

6.2.4 Programmatic Access for IBM Service Information

We can use the **APIClient** class from IBM's Watsonx AI library to programmatically retrieve information about an IBM service, such as foundation models available, see Listing 6.9. We can utilise the credentials created in Figure 6.1 to set the client. The supplementary notebooks provides a comprehensive guide on the in-built functions within **APIClient**.

```
from ibm_watsonx_ai import APIClient

watson_client = APIClient(credentials)

# Example of using the APIClient instance
model_specs = watson_client.foundation_models.get_model_specs()
```

Listing 6.9: Service Information

The method in Listing 6.9 is very useful to check which models are available in watsonx AI and could be used in AI Islands to update the model index due to frequent updates by IBM. However, one should consider how unifying models from other sources could affect the use of such functionality.

6.3 Programmatic Model Inferencing

In this section, we will cover basic model inferencing for both *watsonx AI* and *watson services*. We will reference authentication methods as seen in Section 6.1.

6.3.1 What Is Programmatic Model Inferencing?

Programmatic model inferencing is the process of using AI models through code, often by sending data to an external system like a cloud-based platform, to generate a response. This allows developers to run models in high-performance environments. Inferencing typically occurs via API calls or SDKs, which communicate with models hosted on remote servers or a cloud.

Cloud-based inferencing is particularly important for large-scale AI models, such as our watsonx AI models, which require significant computational power and storage. This enables applications to perform tasks like text generation and classification on demand and without the user required to have excessive computer specifications.

6.3.2 Inferencing Watsonx AI Models

We will start by analysing the inferencing methods for foundation models. *For information on how to inference watsonx embedders, please refer to the supplementary notebooks.*

6.3.2.1 Basic Inferencing

We need to first focus on building an inference connection to the model; sending a request and receiving a response. In order for a successful inference, a user will need to use a correctly configured project ID, see 6.2.2.2. Assuming they have obtained a successful credentials instance (see Listing 6.1), they can execute the following in Listing 6.10.

```
# Import the inferencing class
from ibm_watsonx_ai.foundation_models import ModelInference
# Set the correctly configured project ID
WATSON_STUDIO_PROJECT_ID='your-project-id'

# Create a ModelInference instance with IBMs Granite
inference_model = ModelInference(
                    model_id="ibm/granite-13b-chat-v2",
                    credentials=credentials,
                    project_id=WATSON_STUDIO_PROJECT_ID)
# Use the instance and the built-in generate() to receive a response
inference_model.generate('your-prompt-here')
```

Listing 6.10: Watsonx AI Foundation Model Basic Inferencing

Through experimentation, it was discovered that this is not a sufficient method in itself, as it produces unrelated content with a very small max token size. A solution needed to be investigated in order to use watsonx AI foundation models effectively.

6.3.2.2 Advanced Inferencing

From the previous section, the basic inferencing is clearly not satisfactory, and we need to employ more methods to return any meaningful result. It was discovered that programmatic watsonx AI model inferencing is set up to send a payload within the request, consisting of prompts and parameters, which produce meaningful and accurate results [29]. We will explore model configurations in later chapters. For now, we will describe the basic syntax, as seen in Listing 6.11.

```
from ibm_watsonx_ai.foundation_models.utils.enums import DecodingMethods

# Define the parameters for the model using metanames
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames as GenParams

parameters = {
    # define some parameters...
    GenParams.DECODING_METHOD: DecodingMethods.GREEDY,
    GenParams.MAX_NEW_TOKENS: 100, ...
}

inference_model.generate_text('your-prompt-here', parameters)
```

Listing 6.11: Watsonx AI Foundation Model Advanced Inferencing

We assume the user has already instantiated the inference instance. The general idea here is to set parameters, but this can be improved with the use of setting a decoding method and employing the metanames strategy; using metanames standardises the parameters for different models, helping to avoid errors, which is ideal when working in a larger application.

Through experimentation, it is also advised to set a system prompt (and optionally an example conversation) for the LLM. This can be achieved by modifying the prompt string to include this extra information. Additionally, this is where the idea of inferencing with RAG takes effect, as relevant entries can then be appended to the prompt string. This process also forms the foundation for the chatbot mechanics, which will be explored in more detail in Chapter 8.

6.3.3 Inferencing Watson Services

We will conclude by analysing watson service inferencing, using the NLU as the primary example. *For information on how to inference the other services, please refer to the supplementary notebooks.*

6.3.3.1 Inferencing the NLU Service

Assuming the user has obtained successful NLU credentials via IAM token resource controlling and authenticated with the **IAMAuthenticator** class from IBM's Cloud SDK (see Listing 6.6), they can execute the following in Listing 6.12 using built-in service methods [30].

```
from ibm_watson import NaturalLanguageUnderstandingV1

nlu = NaturalLanguageUnderstandingV1(version, authenticator)
nlu.set_service_url(credentials['url'])

# Now inference nlu with built-in functions
nlu.analyze(text, features).get_result()
```

Listing 6.12: NLU Service Inferencing

6.3.3.2 Inferencing Other Services

We will not cover the inferencing of other services in this report, please see the supplementary material. However, we can essentially follow an almost identical procedure to the NLU set-up. We can use the service specific built-in functions as described in Listing 6.13 below.

```
# Inferencing text to speech
tts.synthesize(text, voice, accept).get_result()
# Inferencing speech to text
stt.recognize(audio, content_type).get_result()
```

Listing 6.13: TTS & STT Service Inferencing

6.4 Watsonx Programmatic Collection Prototype

This section covers the experimental phase results, leading up to *The Watsonx Prototype*. The application is provided within the supplementary materials for this report, mainly for backend analysis.

6.4.1 Prototype Overview

As part of research and experimentation, producing a python-based app containing a collection of watson services available programmatically would prove to be a crucial step before attempting an integration into AI Islands. The app is a collection of experimental tests in earlier stages of the project, showing the development process. We will refer to this application as *The Watsonx Prototype*.

Producing this app served as a guide for the rest of the team producing AI Islands, highlighting how flexible the application needs to be in order to unify different model types from multiple sources. It would also prove to be useful to show stakeholders and external supervisor teams to better understand the project and determine as to which other services could be included.

As seen in Sections 6.1, 6.2 & 6.3 , programmatic access is at the core of the prototype. It features the unified authentication method, utilises resource and account information retrieval, and inferences watson models on a users' cloud account.

6.4.2 Core Features

The Watsonx Prototype showcases IBM Watson's AI tools, and is designed to expose users to their various services. It offers users a system for handling tasks related to text analysis and generation. The app structure allows for seamless integration of multiple IBM AI functionalities, providing useful insights into their products. The core features are listed below:

- **Text Generation**: Supports processing text via AI models.
- **Project and Model Selection**: Allows users to select different projects and models.
- **Prompt Tuning**: Includes customisation of system prompts and example conversations.
- **Parameter Customisation**: Users can adjust various parameters for text generation.
- **Dataset Processing**: Enables uploading and processing datasets for RAG.
- **Retrieval-Augmented Generation (RAG)**: Uses datasets to enhance text generation.
- **Text-to-Speech**: Converts text to speech with various voice options.
- **Speech Recognition (Audio-to-Text)**: Transcribes audio files into text.
- **Natural Language Understanding (NLU)**: Analyses text for sentiment, keywords, etc.
- **Chatbot**: Provides an interactive chatbot interface for conversations.
- **API Key Management**: Users can manage their IBM Cloud API key and location settings.

6.4.3 Extra Features

At this stage of the project and after discovering which IBM services could be made available in a programmatic application, there was exploration of how methods could further be enhanced.

A basic RAG system was integrated into the prototype that provided the user with the ability to upload datasets and provide them as context for their prompts. The basic system used pandas and FAISS, along with an offline sentence transformer.

In another aspect of the main project, an offline chatbot was requested by the stakeholder to enhance the AI Islands application. Building on this, it was decided that integrating a chatbot feature into the watsonx foundation models, along with other offline LLMs, would further unify the final application. As a result, a chatbot is featured in this prototype.

6.4.4 Stakeholder (UAT) Feedback

The feedback from stakeholders concerning *The Watsonx Prototype* was generally positive. They were impressed with how authentication could be unified, how account specific information could be retrieved and how model inferencing could be adjusted by the user.

Watsonx Embedding Models: Discussions took place about the idea of a RAG system in which we explored the use of online watsonx embedders. Having already looked into this, it seemed like a good alternative to the offline embedders and to showcase watsonx even further within a local application.

RAG with Chunking: Discussions about the internal complexities of a RAG system also took place. This is where initial exploration of *dataset chunking* took place and is seen to be later included along with watsonx embedders, within AI Islands as a development.

6.5 Chapter Summary

In this chapter, we analysed the underlying mechanics behind the core of watsonx integration. Authentication was central to unifying access to IBM Cloud services. By implementing a system that used general API keys and IAM tokens, the need for users to manage multiple credentials for various IBM services was eliminated, streamlining the user experience.

Next, we explored information access, focusing on retrieving user account resources and service-specific data. Programmatic access to IBM Cloud services via IAM tokens allowed for validation and resource management, ensuring users had the correct account configurations.

We also analysed model inferencing, covering both watsonx AI models and watson services. We learnt the core aspects of programmatic inferencing of LLMs, with advanced techniques for altering prompts and tuning parameters to improve outcomes.

Lastly, *The Watsonx Prototype* was an experimental tool that provided valuable feedback, helping guide the integration of IBM services into AI Islands and shaping its final design.

Chapter 7

Implementation Part II: AI Islands Watsonx Integration

This chapter signals the start of the watsonx integration implementation into AI Islands. References to the watson technical research implementation chapter will be made throughout. Additionally, it is recommended the reader now reference the AI Islands source code within the supplementary material provided.

7.1 AI Islands - IBM Integration

In this section, we will cover the main integration points between AI Islands and IBM Watson. We will start with the backend implementation by analysing the interaction between the control classes and the watson specific classes along with validation, authentication, and RAG utilities. We will then look at the frontend integration (meaning the frontend required for IBM services to operate) and finally cover the improvements from *The Watsonx Prototype*.

7.1.1 Backend Integration

The reader should now reference Figure 5.1. We will analyse the mechanics of the backend that is allowing watson services to operate within AI Islands. The main structure consists of control classes called **ModelControl**, **LibraryControl**, and **PlaygroundControl**. In this chapter, we will primarily refer to **ModelControl**; with the other classes referenced in later analysis.

Every model type within AI Islands is associated with a model class. The structure allows for expandability; if a developer wishes to include models from another source, they can simply create a new model class, provided it follows the *BaseModel* structure. In the case of AI Islands, this means that a new model class is required to have a *download()*, *load()* and *inference()* function. It is essentially set up for offline models that require an actual download, but we can think about what a download and load could mean for a cloud-based run model and how we could include some preliminary model set-up functionality instead of trivially passing the function, see Section 7.1.1.5.

7.1.1.1 Watson Classes

From previous chapters, it is clear that there is a distinction between the set up for both *Watsonx AI* and *Watson Services*. The decision was made to create two classes, with **WatsonModel** and **WatsonService** representing these; assume for a moment that IBM decide to make more services available, then using one class will become very large and complex, and in such cases, one must consider future expandability.

7.1.1.2 Authentication Classes

AI Islands uses three classes in one authentication file to carry out authentication operations, defined as Authentication, ResourceService, and AccountInfo. Authentication validates API keys, validates user projects, and fetches the user IAM token from an API key. The ResourceService class fetches the user resource list and service credentials based on the IAM token retrieved from the Authentication class method. Lastly, AccountInfo combines these methods and creates credentials and a watson API client, aswell as providing a users project list. We will abstract this away into a collection called the *AuthenticationClasses* when referring to watson authentication methods. *In the AI Islands source code, these classes can be found in the directory* `backend/utils`*, within the file* `ibm_cloud_account_auth.py`*.*

7.1.1.3 Settings Classes

AI Islands uses a SettingsService and WatsonSettingsManager class to configure user settings. The SettingsService controls the overall settings to include environment variables (related to watson including API key, location and project ID) and system settings (related to RAG and Hardware). The WatsonSettingsManager is more specialised in that it controls the environment variable file, in which the code is very intricate in order to reset environment variables within the same session. This allows the user to update their environment variables on the fly, without needing an application restart. *In the AI Islands source code, these classes can be found in* `settings_service.py` *within the directory* `backend/settings` *and* `watson_settings_manager.py` *within the the directory* `backend/utils`*.*

7.1.1.4 RAG System Utility Classes

The AI Islands RAG system consists of two main classes; a dataset controlling class called DatasetFileManagement, and a data science class called DatasetManagement. The DatasetFileManagement is mainly responsible for uploading, previewing, and retrieving processing information for the user, and consists of other similar methods. The DatasetManagement class is responsible for initialising embedding models, generating embeddings, processing datasets, and similarity searching (in particular containing the find relevant entries RAG function). We will re-visit the intricacies of the data science in Chapter 8. *In the AI Islands source code, these classes can be found in the directory* `backend/utils`*, within the files* `dataset_management.py` *and* `dataset_utility.py`*.*

7.1.1.5 Download, Load, and Inference Methods

As previously stated in section 7.1.1, the WatsonModel and WatsonService classes must follow the base model structure to contain the *download()*, *load()* and *inference()* methods - *the base model is located in the* `backend/models` *directory*. Detailed analysis is provided below, explaining the intricacies of these methods within the watson classes. *In the AI Islands source code, these classes can be found in the directory* `backend/models`*, within the files* `watson_model.py` *and* `watson_service.py`*.*

Download: This method will produce a library entry within AI Islands. For the watson classes, *download()* validates the user general cloud API key and checks if the necessary resources have been added to the users' account. It does this by using a method within *AuthenticationClasses* which fetches the user resource list and *download()* will compare the data received; if the user is missing a resource they will be notified of which one(s) through the application.

Load: This method will load the model into a child process, and in the case for the watson classes, we can initialise the services here. This ensures that once the model is successfully loaded, it is ready for the user to begin inferencing. The *load()* method starts by validating the API key as before, in case the user changes it. From this point on, the *load()* method is slightly different between WatsonModel and WatsonService, based on their differing requirements.

(Case: WatsonModel) After API key validation, *load()* tries to fetch the user project ID from the environment variables. If there is no project, the function will employ a method from *AuthenticationClasses* that programmatically fetches an ID from the user account - making AI Islands more user-friendly. If there is a project ID present, it will be validated through *AuthenticationClasses*. If validation for both API key and project ID passes, then *load()* will either create an instance of the ModelInference class from the IBM Watsonx AI library if the model is a foundation model (see 6.3.2), or it will create an instance of the WatsonxEmbeddings class from the LangChain IBM Library if the model is an embedding model.

(Case: WatsonService) After API key validation, *load()* fetches the service name from the index. The service name is used to determine which service to initialise, and this is done through the use of private methods as seen in 6.3.3 which employ methods from *AuthenticationClasses*, which we have analysed in Section 6.1. If successful, *load()* will return the authorised service instance.

In **both** cases, load() returns an **authorised instance** of the model set-up, ready for user inferencing.

Inference: This method will inference the authorised instances. In **Section 6.3**, we have seen how this can be accomplished, and the methods employed here are very similar - we define the necessary parameters and utilise the model handler's specific built-in functions.

However, we can expand on this in the case of the watsonx foundation models. In 6.3, we briefly mentioned that the user can modify the prompt string, and that this is where dataset entries are appended within the payload. We can also append the chat history to the prompt string in the case of the chat bot feature (explained in more detail in Chapter 8).

Configurations will be discussed in Section 7.2, but for watsonx foundation models we have four categories of configurations used within inferencing; Prompt, Parameters, RAG, and Chatbot. It is important to note that all of these are set and used within this *inference()* method.

7.1.2 Frontend Integration

The reader should now reference Figure 5.2. Generally, it should be noted that the FastAPI is enabling the frontend and backend to communicate using the backend **Router** and frontend **Service** classes only. This means that when integrating watsonx models into AI Islands, the frontend can be generalised to be applied to all models. Providing the frontend inference UI can change interface, we can use any model provided in the AI model index.

However, we need to make clear to the user what the difference is between offline and online watsonx models. We do this by using a status tag, defined within the model index and library listing pages. We also include the necessary logic to allow backend exception handling messages to show through the UI to inform users of issues regarding watsonx model interaction.

7.1.2.1 Watson Targeted Frontend

In order to fully integrate watsonx services into AI Islands, we need to ensure the necessary UI pages are included for watsonx-exclusive features. The main page necessary is for the watsonx IBM cloud settings. The user needs somewhere to enter and update their API key, location, and project ID otherwise they cannot run any IBM services.

Additionally, the online LLM models use a RAG system for context-based prompting and so a data refinery UI page is needed in order for the user to create and process their datasets.

7.2 AI Islands Core Features

This section covers the core features related to IBM watson services in AI Islands. Please see **Appendix E** for the full set of watson models integrated into AI Islands.

7.2.1 IBM Watson Model Index

In the previous section, we analysed in detail how watsonx models are implemented within AI Islands. We can now give a more general implementation, now that we have this knowledge. As stated in previous chapters, the model index consists of a variety of offline and online models, in which all of the online models are provided by IBM watsonx.

7.2.1.1 Foundation Models

AI Islands currently features 15 different watsonx AI foundation models, including IBM's Granite. Foundation models can be downloaded from the AI Islands Model Index by either browsing or using the search and filter mechanism (one can filter for type set to text-generation, and status set to online). Once downloaded, a user can load them into memory. The user can also view information about the model, inference, configure, and use the API access instruction tabs. If the chat history feature is turned on, then the models' inference page will turn into a chatbot view.

Configuration: Foundation models each have the same configuration set up consisting of four configuration types; prompt, parameters, RAG, and chatbot. We have already analysed prompts and parameters in Chapter 6 and will analyse RAG and chatbot in Chapter 8. For now, please see Appendix E for the full set of configurations and their definitions.

7.2.1.2 Embedding Models

AI Islands currently features 6 different embedding models. Embedding models are available through the AI Islands Model Index, and have their associated inference and API access pages. A user can inference the models but this will just produce a vector embedding. The main purpose of having this model present within the index is for education purposes (to see what it does) and mainly for API access if a developer wishes to use the model from an external application. Embedding models are also featured within the RAG utility, but this will be covered in later chapters.

Configuration: At this time, the embedding models are not configurable in the backend and operate at a default value. However, in future work there can be exploration of this feature.

7.2.1.3 Natural Language Understanding (NLU) Service

The NLU is available in the AI Islands Model Index under a text-classification model. The service is treated as just another model, and features the usual detail, inference, configuration, and API access pages. When a user inferences with text, the response is given as readable key-value pairs, with the option of a JSON view to show instead.

Configuration: Users have the option to hide certain features by checking options.

7.2.1.4 Text to Speech (TTS) Service

Similar to the NLU, the TTS is accessible in the index and has the same associated pages. However, the output is audio and instead of a standard text box, we have an audio player. When inferenced, the audio player will update with new responses and can be played multiple times; this is very useful in case the user misheard the first time!

Configuration: Users have the option to select from 52 different voices, pitch and speed.

7.2.1.5 Speech to Text (STT) Service

Similar to the TTS, the STT is accessible in the index and has the same associated pages. However, in this case the input is an audio file upload, and so the input frame changes to account for this.

Configuration: Users have the option to select from 86 different transcribing models.

7.2.2 Model Customisation

In 7.2.1, we have looked at the individual configuration types for each model. Any model's customisation functionality is identical; users can restore from default, save, and save under a new model with modified configurations. There are many advantages for this:

- **Input Acceptance**: Providing a system prompt and example conversation to better understand a different models' output can be beneficial for use within the playground (see 7.2.3).
- **Chatbot**: Turning chat history configuration on, and saving the model under a new name for quick access to this popular feature.
- **Imposing Cost Limitations**: The user can customise a model for shorter responses by adjusting parameters and prompt settings, ensuring clear identification.
- **Associated Dataset**: The user can set up a model to be configured to work with a dataset through the RAG feature, and can name this new model under a more obvious name.

7.2.3 Model Chaining

AI Islands features a playground which consists of similar pages to the library set, see Figure 4.8. The user can create, add models, and configure the chain order within a playground. They are then able to inference the playground model chain; the original input is fed through the first model, which provides an output, which in turn is fed through to the second model, which provides a new output, until this iteration is complete and the user is provided with the final output of the chain.

This is particularly useful for creating new purpose models while still maintaining the original base models. In the last section about configurations, we briefly mentioned input acceptance and how this can be used to change the model behaviour. We can best describe chaining with a particular use case. Take the audio image description system for visually impaired people, as described below.

Audio Image Description System: The user can create this system by searching the model index for an object-detection, text-generation, and text-to-speech model. They choose an offline computer vision model (yolov10s) and download this; meaning the actual model will be downloaded to their project directory. The user decides they want an online watsonx text-generation (ibm/granite-13b-chat-v2) and an online TTS (ibm/text-to-speech) in which they download; meaning API key validation takes place. For the Granite model, the user decides to strengthen its input acceptance by altering its prompt configuration with a new system prompt and an example of yolov10s output.

Once all models are prepared within AI Islands, the user can create a playground titled 'Audio Image Description System', and add the downloaded models, while configuring the model order within the chain ordering tab; in the order yolov10s, ibm/granite-13b-chat-v2, ibm/text-to-speech. The user can now load the chain and inference by uploading an image input and submitting. The user will receive audio containing information about the objects detected within the initial picture.

7.2.4 Data Processing

Users can process datasets for RAG within AI Islands' data management tab. The application offers users the ability to upload their own datasets in a variety of formats (see Appendix E.3) and for different purposes. The user is presented with many embedding model choices of which the embedder is either offline or online, as defined in E.3.

Once processed via *default* or *chunked* method, the RAG files are created within the dataset specific method directory. The user can view a processing summary and report for each processing method provided they carried out either one. The useful point here is that users can re-visit the data refinery, select the dataset, and view its processing information from before. If the user re-processes, then the method folder will be overwritten and new reports generated.

The user can configure an LLM to use the RAG files for its context. For more details on the processing and RAG, please read Chapter 8.

7.2.5 Settings

Users have three main types of settings; IBM Cloud Account, RAG Settings, and Hardware Settings. Each contain their own setting options as shown in Appendix E.4. We will focus on the first two.

IBM Cloud Account: These settings allow the user to edit their API key, location, and project ID. As mentioned in Chapter 6, all of these are essential for running watson services. We have already explained the intricacies of unifying authentication, highlighting the user-friendliness of entering less information in order for the user to run watson services in AI Islands.

RAG Settings: These settings control the dataset processing stage within the data refinery, and are treated as global settings. If a user wishes to alter the way the processing occurs, or even change the method from *default* to *chunking* and vice versa, they can.

7.2.6 Improvements from The Watsonx Prototype

AI Islands offers many improvements from the prototype. We now have many different types of offline models available to the user, and we have the addition of watsonx AI embedding models within the model index and the data processing stage. Some of the original IBM services are more enhanced; the IBM Speech to Text service now contains many models, expanding the user choice.

Users now have the playground model chaining functionality, in which they can experiment with combining many models together from the model index. Models have more complex configurations, further enhancing their capabilities.

In particular, the data science within the RAG system has improved drastically, to allow for user choice in how documents are processed with different methods appropriate for their use cases.

7.3 Frontend Coding

In this section, we provide an overview of the frontend development. While significant time and effort were dedicated to this aspect, it is not the primary focus of the report, so only a high-level summary will be presented, along with decisions taken to overcome any issues. For the technical reader, they should refer to the material in **Appendix F**.

In this section, we will cover the main process in building the UI pages. The reader should reference Figure 5.2 which shows the structure of a .NET MAUI app.

7.3.1 App Shell & Navigation Bars

As a constant UI feature on all pages, it was vital that the proportions of the navigation bar and its contents were optimal for a more professional appearance; for example, the navigation bar contains both logo and image, whereas the top bar contains them separately. The app shell required some iteration due to a built-in bug concerning the three dotted drop down menu. It is a built-in toolbar that cannot be removed and instead of designing a custom shell, it was populated with options such as links to model source websites as a work-around. For the nested navigation bars present in the library, playground and data management pages, referred to as *tabbed pages*, it was important that the bars had the same design.

7.3.2 Index Pages

The index pages consist of the Model Index and the Library, in which there is a pop up filter. The filter is particularly good for future expandability when more models are added to the index. For the Library, we can delete, load, and unload all models; the 'unload all' is particularly useful for automated model unloading from their respective child processes. Users can also compare the model with the base model, in case of a customisation occurrence, and can see if a model has a different configuration, through the custom column.

A loading / unloading indicator along with an inactive background setting was implemented for a better UI design.

7.3.3 Inference

Inferencing is a particularly challenging UI page to produce. We need to ensure all model types can function within this page; an object-detection needs a file uploader as input, a text-generation needs a text box, and a text to speech needs an audio player output. Additionally, we should promote user-friendliness by formatting the model output, providing previews of images or video inputs, and providing an option for a JSON toggle view for the technical user.

For more information on the implementation of inference, see **Appendix F**

7.3.4 Configuration

Similar to the inference page challenges, we need to ensure that any type of configuration present within a model's configuration section is shown to the user through the UI. When we consider the vast amount of configurations possible, this poses a great challenge. Several methods were explored, such as generating the UI through the logic of dictionary degrees of nestedness which is very flexible when it comes to unknown configurations, but eventually it was decided to use visibility binding to show if a setting existed within a model's configuration. This allowed for easier UI design development for particular configuration sections.

In terms of the user experience, expandable views for configuration sections were used, which are particularly useful for models with lengthy configurations. Buttons featuring resets and saves were also implemented, with frontend error and navigation handling. Slide bars were utilised when appropriate which presents the user with minimum, maximum and optimised values (in green) for a particular configuration; research was conducted into this to determine the exact ranges and optimisations.

For more information on the implementation of configuration, see **Appendix F**

7.3.5 Data Management

Data management contains the data refinery, which is essential for processing datasets for use in an LLM's RAG feature. Users can upload their own datasets or remove them completely. The user can then select a dataset, preview, and process it. The frontend prompts the user to select an embedding type first, before presenting them with the precise embedder. This separates the clear distinction between the offline and the watsonx online embedders. Once a processing configuration is selected, the process button turns from red to green, indicating to the user that the processing selection is valid.

Selecting a dataset will provide the processing reports for both default and chunked provided they were processed with either method, which ensures that the user can keep track of the processed dataset easily.

7.3.6 Settings

The settings page is fairly basic in nature, we have three sections; IBM Cloud, RAG, and Hardware. The IBM Cloud API key is masked for stronger security. The RAG settings are appropriately frontend validated; the user can save any of the options and it will not affect the backend functionality, but the use of frontend combinatory validation is used to instruct the user as to which setting is being used for the chunk method selected. Additionally, the hardware choice between cpu or gpu is validated so that selecting gpu with a computer that does not meet the requirements will automatically return to the cpu option.

Moreover, the user is presented with '?' buttons, which provide information to the user to better understand the functionality of each setting option.

7.4 Chapter Summary

In this chapter, we analysed the main integration of AI Islands with watsonx, focusing on both backend and frontend aspects. For the backend, we explained how control classes are managing the watson-specific classes, which handle watsonx AI models and watson services, while authentication and RAG functionalities were produced as utilities for the model's life-cycle. The core watson model-service methods of download, load, and inference, were designed to allow similar interactions between models and services.

The chapter also outlined AI Islands' core features, including a comprehensive model index that in particular supports foundation models, embedding models, and watson services. The playground functionality enables users to create model chains with watsonx online models, offline models, or a mix of them. Additionally, the data processing capabilities, particularly the RAG system, were highlighted for their importance in customising model outputs.

In the Frontend Development section, we discussed the UI's development, focusing on user-friendly navigation, model index pages, and inference setups that cater to different input-output types. We covered the complexity of configuration pages, the data management system for processing datasets, and the settings page that manages IBM Cloud API keys, RAG options, and hardware choices. The frontend ensures a smooth interaction between users and the complex backend functionality.

Chapter 8

Implementation Part III: Enhancing Watsonx

This final implementation chapter explores the enhancement of watsonx within AI Islands, through the Retrieval-Augmented Generation (RAG) and Chatbot functionalities. *For the technical reader, the AI Islands source code is provided, and should be referenced where appropriate.*

8.1 Context-Based Prompting

This section covers context-based prompting with LLMs, addressing prompt size limitations and introducing RAG to filter relevant data. It also explores using chatbot features to retain conversation history and improve watsonx models by combining RAG and chat history for better accuracy.

8.1.1 Prompting

Prompting is a method that gives an AI model specific instructions to guide its response, and it is the primary way a user interacts with AI systems. The accuracy of the response can change given the input provided to it by the user [31]. In this chapter, we will only consider prompting with LLMs.

8.1.2 Limitations

Prompting can have its limitations. In the case of watson services where requests are sent via API to the cloud for processing, there is usually an input token limit, known as a *context window* [32]. This means a user cannot provide greater context in order to produce an accurate result. We need a method that can take large prompts and filter them in order to reduce the prompt size while still maintaining all of the relevant information to produce the accurate response [33].

8.1.3 Retrieval-Augmented Generation (RAG)

RAG is a method where an AI system looks up relevant information (from a database or document) related to an initial user prompt, then uses that information to generate a more accurate response. As opposed to just relying on previous knowledge, a RAG system helps the LLM accumulate particular details it needs, making the response more accurate [34]. It is especially useful when the LLM needs to answer questions or provide information that might not be part of its training data.

In the context of large datasets, we can use a RAG system to filter out relevant entries, thus decreasing our token input while still maintaining the necessary context for an accurate LLM response [35]. Alternatively, we could focus on methods which appear to increase the context window size,

possibly by chunking larger prompt requests, but sometimes providing too much context can 'cloud' the context base, meaning the response accuracy diminishes [34]. With this 'less is more' approach, we can implement a RAG system within AI Islands, whilst considering this alternative as a valid option within the RAG system itself (see the *default* method in 8.2.4).

8.1.4 Chatbot

Now assume we want to have a meaningful conversation with an LLM without the user being required to enter information provided by an LLM regarding each response. We need a method that keeps the prompt context for each inference. In other words, we need to provide the LLM prompt and response history as context for the next prompt, see Figure 8.1 below.

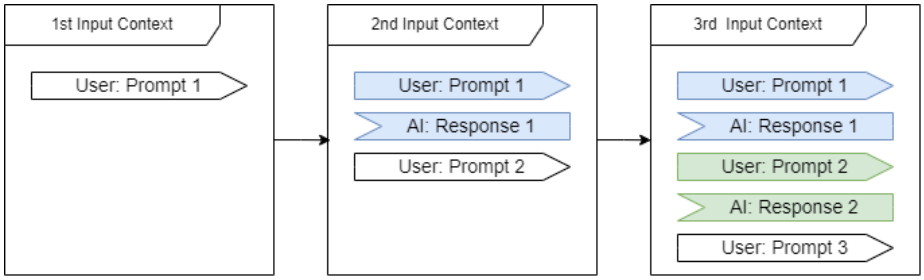


Figure 8.1: Chatbot History Mechanics Diagram

This chatbot feature can easily be incorporated into the *inference()* method for the LLM classes, as seen in Listing 8.1. We can initialise the model class with an empty list that will hold the conversation history. During inference, the current conversation list is passed to the LLM, which uses it to provide context for generating a response. After each inference, we append the prompt-response pair to the list, ensuring the conversation context is maintained for the next inference.

```
class WatsonModel(BaseModel):
    def __init__(self, model_id: str):
        ...
        self.chat_history = [] # initialised as an empty list

    def inference(self, data: dict):
        ...
        # during inference, provide all history
        if use_chat_history:
            for message in self.chat_history:
                full_prompt += f"{message['role']: message['content']}\n"

        # after inferencing, new history is added to the list
        if use_chat_history:
            self.chat_history.append({"role": "human", "content": payload})
            self.chat_history.append({"role": "ai", "content": final_result})
```

Listing 8.1: Chatbot Functionality Components

8.1.5 Enhancing Watsonx LLMs

We can use a combination of RAG and Chatbot features to enhance watsonx foundation models. We can modify the prompt string when inferencing the model by appending relevant entries (for RAG) or by appending the full chat history every time the LLM is inferenced. These modifications occur within the **WatsonModel** *inference()* method, as shown with the chatbot feature in Listing 8.1. Now, we know that IBM's models have a restriction on token input limits (i.e. prompt size) and so by adopting a RAG system, we can enhance the programmatic use of watsonx LLMs.

8.2 Data Processing

This section covers data processing in AI Islands, while describing the two methods *default* and *chunked*. We finish the section with dataset storage and offline LLM integration concerns.

For the technical reader, the supplementary notebooks should be referenced. In the AI Islands source code, the methods discussed in this section can be found in the directory `backend/utils`*, within the files* `dataset_management.py` *and* `dataset_utility.py`*.*

8.2.1 Embedding Models (EMs)

Embedding models (EMs) are models that convert text into embeddings. In other words, providing a text prompt to an EM will produce a vector of floats. AI Islands contains two EM types; sentence transformer models from the sentence transformer library that run offline, and watsonx embedders that are online and run on the cloud, requiring similar access as to the watsonx foundation models. This presents the user with a choice and further highlights the watsonx integration into AI Islands, and the *islands* theme itself.

Using watsonx EMs programmatically is a very similar process to using watsonx foundation models as seen in Listings 6.1, 6.10 & 6.11, and so we will not cover the techniques in this chapter. However, for the technical reader, the supplementary notebooks explores watsonx EMs.

8.2.2 The AI Islands RAG Concept

In order for RAG to work, datasets need to first be prepared and processed into a vector embedding format (referred to as a vector database). Assuming a dataset has been processed in this way, we would have a database consisting of vectors instead of human readable text. We can take the users' initial text prompt, convert it into a vector embedding and compare how similar this vector is to all of the vectors within the vector database using similarity searching. An LLM is then prompted with the initial text prompt appended with the relevant entries in text format.

The supplementary notebooks explores the RAG concept using a fictional dataset as context with watsonx LLMs in two ways: either with an offline sentence transformer or an online watsonx embedder. Technical readers are encouraged to review these examples before continuing.

8.2.3 Dataset Processing

The dataset processing functionality in AI Islands provides a pipeline for preparing data for RAG. It begins by initialising the appropriate embedding model, either a watsonx embedder or sentence transformer, based on user choice. The process then handles various file types, with special consideration for CSV files. For text data, it applies configurable chunking strategies (fixed length, sentence-based, or paragraph-based) if enabled.

The core of the processing involves generating embeddings for each text chunk or entry using the initialised model. These embeddings, along with the original text data, are then used to create a FAISS index for efficient similarity search. The procedure then saves all processed data, including the embeddings and FAISS index, organising them in a structured directory in `Datasets/`. Throughout the process, it manages metadata and generates detailed reports, providing insights into the dataset characteristics and the processing outcomes.

Within AI Islands, users can process datasets in two different ways called default and chunked. Both are available for the same dataset and users will have the configuration option to use either method folder when inferencing an LLM, provided they had processed the dataset under both methods. If the user re-processes via either method again, and perhaps with a different EM selected, both default and chunked folders will be overwritten, while updating the model tracker and generated reports.

In the AI Islands source code, the main method process_dataset(), within DatasetManagement, can be found in `dataset_utility.py`. The method utilises multiple private methods in this class for different processing types, which we will now analyse in 8.2.4 & 8.2.5.

8.2.4 AI Islands Default Processing

If the user processes a dataset through default processing, then no chunking will be applied. This means that each entry in the dataset is treated as a single unit for embedding and retrieval. CSV data files will embed per data row, and documents will embed as one single embedding. This approach preserves the full context of each entry but may lead to less fine-grained search results compared to chunked data, especially for longer texts.

Preserving entire documents can offer benefits particularly for offline LLMs that can handle large contexts. This method maintains full contextual integrity, crucial for tasks needing holistic understanding like legal document analysis or complex research queries. While it may provide less specific search results than chunked data, it excels in capturing long-range dependencies and complex relationships within texts. This approach is particularly valuable when document-wide context is essential for accurate interpretation and when working with LLMs capable of processing and leveraging extensive context windows effectively. However, we can still achieve preservation of large context windows with chunking; the user can adjust the settings to their needs. However, the default method can offer a quick and easy way for context-based prompting for users with less knowledge of the intricacies of RAG mechanics.

For online LLMs using cloud APIs, the default approach can pose some challenges. API token limits, higher costs, increased latency, and potential loss of relevance precision are key issues. These LLMs often benefit more from chunked data, which allows efficient use of token limits, reduces costs, improves response times, and enables more focused answers. However, this comes at the cost of losing the broader document context that the non-chunked method preserves.

8.2.5 AI Islands Chunked Processing

If the user processes a dataset through chunked processing, the **DatasetManagement** class applies text segmentation based on the specified chunking method. The implementation supports several chunking strategies:

- **Fixed-length:** splits text into chunks of a specified size, with an option for overlap.
- **Sentence-based:** divides text at sentence boundaries, using periods, question marks, and exclamation points as delimiters. It combines sentences up to a specified chunk size, with a specified overlap determining the number of sentences shared between adjacent chunks.
- **Paragraph-based:** splits text at paragraph breaks, defined by double newlines. Similar to sentence-based, it combines paragraphs up to a specified chunk size, with a specified overlap specifying the number of paragraphs shared between chunks.
- **CSV row-based:** allows combining multiple rows into a single chunk, with options to specify which columns to include. This method does not use a specified chunk size or chunk overlap.

Each chunk is embedded separately, resulting in multiple embeddings per document or data entry. Additionally, the original chunks, along with their embeddings and the FAISS index, are saved for later retrieval. This approach allows for more granular representation of the content, enabling finer-grained search and retrieval capabilities.

The chunked approach offers flexibility in handling large documents. It allows for more precise matching of specific sections, which can be beneficial for targeted information retrieval. The granularity of chunks enables the LLM to focus on relevant portions of text, potentially improving the accuracy of responses to specific queries. However, the effectiveness depends on the chosen chunk size and overlap, which users can adjust to balance between preserving local context and enabling specific matching. For online watsonx LLMs, the chunking approach effectively manages token limits by processing smaller chunks instead of entire documents per API call. This allows more efficient use of the model's context window, enabling longer document processing by sending chunks instead.

The chunking approach provides a balance between preserving context and enabling targeted matching. Users can adjust chunking parameters through RAG settings to optimise for their specific use case, whether working with offline LLMs that can handle larger contexts or online LLMs with API and token constraints. The implementation also includes visualisation and reporting features, generating processing reports with chunk distribution plots, which can help users understand and optimise their chunking strategy.

8.2.6 Dataset Storage

Through research, IBMs Cloud Object Storage (COS) seemed to be an alternative for storing datasets, as opposed to locally. We can successfully upload processed datasets to the COS, as seen in the supplementary notebooks. However, when it comes to uploading a dataset and processing it programmatically, this proved to be too complex and required the user to interact with the IBM Cloud platform. It is equally as challenging to utilise the processed data on the cloud when inferencing LLMs with RAG.

However, when it came to unifying the system, it was decided to keep these processed datasets locally. This decision was made since dataset management is a separate entity within AI Islands and benefits the implementation of RAG for offline LLMs under future work.

The supplementary material explores COS data storage using a fictional dataset, which technical readers can optionally review as a potential area for future development.

8.3 AI Islands RAG System

This section covers RAG application to LLMs in AI Islands, while referencing the previous sections as context. We conclude by explaining the application to the model class inference method.

8.3.1 Similarity Searching

The similarity between the initial prompt vector and the individual entries in the vector database can be measured in different ways, such as using cosine similarity or euclidean distance. We can use a model that specialises in similarity search. We use FAISS for this task within AI Islands, since it is preferred for its speed, scalability, and future expandability. As datasets grow or new models are integrated, FAISS's support for both CPU and GPU environments ensures it can handle increased complexity. Unlike simpler libraries such as Annoy or Scikit-learn, FAISS has robust indexing structures, allowing for efficient handling of high-dimensional data. This flexibility and advanced indexing make FAISS more adaptable for evolving system requirements, positioning it as a longer-term solution for managing large-scale similarity search efficiently.

The similarity searching functionality in AI Islands uses a processed dataset to find relevant entries for given queries. It utilises the FAISS index created during the datasets' own processing stage, which allows for fast and efficient similarity comparisons. When a query is received, **DatasetManagement**'s *find_relevant_entries()* method first generates an embedding for the query using the same model used for dataset processing. It then performs a similarity search against all vectors in the FAISS index, retrieving entries that exceed a specified similarity threshold. The search process can work with both chunked and non-chunked data, adapting its approach based on how the dataset was initially processed. Retrieved entries are ranked by similarity score and formatted for output. This method enables quick and accurate retrieval of relevant information from large datasets.

8.3.2 Data Embedding Consistency

Upon prompting an LLM with a query in a RAG system, the query is converted to an embedding in order to perform the similarity searching. However, we need to ensure that the EM used is identical to the EM that was used for processing the dataset in the first place. We can do this by producing a JSON file during processing which keeps track of the EM used. We can then access this same JSON information later for when the user decides to use RAG datasets within the LLMs. Embedding model consistency is important because of the following:

- **Consistency in vector space**: Different embedding models create different vector spaces. By using the same embedder, we ensure that the query vector and the dataset vectors exist in the same high-dimensional space, making their comparison meaningful.
- **Dimensionality matching**: Each embedding model produces vectors of a specific dimension. The FAISS index is built to work with vectors of the same dimension as the dataset embeddings. Using the same embedder ensures that the query vector has the correct dimensionality to be compared with the indexed vectors.
- **Feature representation**: Each embedding model captures different aspects of the text in its vector representation. Using the same model ensures that the same features are being compared between the query and the dataset entries.
- **Semantic consistency**: The semantic meaning captured by different models can vary. Using the same model ensures that the semantic relationships captured in the dataset embeddings are consistent with those in the query embedding.
- **Performance optimisation**: FAISS and similar indexing structures are optimised for comparing vectors from the same distribution. Using the same embedder helps maintain this distribution consistency.

8.3.3 Similarity Threshold

Users can adjust the similarity threshold configuration on their LLMs to pick up more or less relevant data in the vector database search. The threshold, typically set between 0 and 1, determines the minimum similarity score required for a chunk or document to be considered relevant to a given query. After generating embeddings for the query and comparing them to the stored embeddings using the FAISS index, the method filters the results based on this threshold. Only entries with similarity scores exceeding this value are returned as relevant.

This mechanism allows for fine-tuning the balance between recall and precision in search results. A lower threshold will return more results, potentially including less relevant ones, while a higher threshold will be more selective, returning only highly similar entries. Users can adjust this threshold through the LLM's configuration to achieve the desired level of relevance or to obtain a broader or more focused set of results.

8.3.4 RAG Application

The **WatsonModel** class implements a (RAG) approach, leveraging the *find_relevant_entries()* method from **DatasetManagement** to enhance prompt construction. When RAG is enabled, the inference method in **WatsonModel** extracts relevant settings and utilises **DatasetManagement** to search for semantically similar entries in the specified dataset. This search process employs a FAISS index for efficient similarity comparisons, considering factors such as chunking preference and a defined similarity threshold.

If relevant entries are identified, they are seamlessly integrated into the prompt under a ”Relevant information:” section, effectively augmenting the original user input with pertinent context. This enhanced prompt, combining the user's query with relevant dataset information, is then processed by the LLM as seen in Listing 8.2. This approach allows for more informed and contextually rich responses, particularly beneficial for queries that require specific knowledge from the dataset.

```
class WatsonModel(BaseModel):
    def __init__(self, model_id: str):
        ...
    def inference(self, data: dict):
        # retrieve the current RAG configuration for the LLM
        rag_settings = self.config.get("rag_settings")

        if rag_settings.get("use_dataset"):
            # define configuration variables for RAG function parameters
            dataset_name = rag_settings.get("dataset_name")
            similarity_threshold = rag_settings.get("similarity_threshold")
            use_chunking = rag_settings.get("use_chunking")

            if dataset_name:
                # create an instance of the data utility class
                dataset_management = DatasetManagement()
                # call the RAG function and store the entries
                relevant_entries = dataset_management.find_relevant_entries(
                    payload,
                    dataset_name,
                    use_chunking=use_chunking,
                    similarity_threshold=similarity_threshold
                )

                # construct the prompt with these fetched data entries
                if relevant_entries:
                    full_prompt += "Relevant information:\n"
                    for entry in relevant_entries:
                        full_prompt += f"- {entry}\n"

        # inference as normal, but with this augmented prompt
        result = self.model_inference.generate_text(full_prompt, params)
```

Listing 8.2: RAG Functionality Components

The system's flexibility in adjusting similarity thresholds and chunking strategies enables fine-tuning of the RAG process, striking a balance between providing comprehensive context and maintaining focus on the original query. This integration of RAG capabilities into an LLM prompt construction exemplifies how retrieval mechanisms can be effectively combined with generative models to produce more accurate and informative outputs.

8.4 Chapter Summary

In this chapter, we explored context-based prompting with LLMs, focusing on overcoming prompt size limitations and improving response accuracy using Retrieval-Augmented Generation (RAG) and chatbot history. We discussed how prompting serves as the primary method for interacting with AI, but it faces constraints like limited context windows, especially in cloud-based systems like watsonx. RAG was introduced as a solution, allowing for relevant information retrieval from datasets to maintain a manageable input size while improving response quality. The chatbot feature ensures conversational context retention by passing previous interactions to the model, which enhances the user experience by maintaining a coherent dialogue.

The chapter also examined data processing techniques within AI Islands, specifically the *default* and *chunked* methods. Both methods serve different purposes depending on whether the LLM operates offline or online, balancing context preservation and token limits for efficient retrieval and accurate LLM responses.

Finally, the chapter delved into the RAG system in AI Islands, highlighting the process of embedding datasets, performing similarity searches, and using FAISS for efficient vector comparisons. The integration of RAG within the LLM inference in AI Islands was demonstrated through the WatsonModel class, showing how relevant dataset entries can be appended to prompts to enhance context and output accuracy. This approach combines LLMs with retrieval mechanisms, thereby enhancing LLMs for a more precise and informed AI responses.

Chapter 9

Testing

9.1 Code Testing

AI Islands code testing mainly consisted of Unit and Integration API endpoint testing due to the FastAPI set-up adopted within the backend structure. Frontend code testing will not be covered in this report.

9.1.1 Unit Testing

We can test the individual 'units' of each of the main class functions that are directly tied to AI Islands watsonx integration. We have sixteen test files which are grouped into five test folders that test the main class categories as follows:

- Watson Model Functions: `models/watson_model`
- Watson Service Functions: `models/watson_service`
- IBM Cloud Account Interaction Functions: `utils/IBMCloudAuth`
- RAG Dataset Functions: `utils/RAG`
- Settings Functions: `utils/Settings`

For Watson Model Functions, tests ensure proper initialisation, downloading, loading, and inference for the WatsonModel class (responsible for watsonx AI models), including API key handling, chat history, and error cases. Likewise, for Watson Service Functions, tests cover initialisation, download, loading, and inference for the WatsonService class (responsible for the NLU, TTS, and STT services), ensuring interaction with APIs and handling errors.

Tests in the IBM Cloud Account Interaction Functions category focus on Authentication, ResourceService & AccountInfo class methods, along with project handling functions, verifying API key validation and proper API interactions. For RAG Dataset Functions, tests focus on DatasetFileManagement & DatasetManagement methods, validating dataset management, embedding generation, and retrieval of relevant entries, using mocks to simulate external dependencies. Finally, the Settings Functions category, tests SettingsService & WatsonSettingsManager class methods, ensuring correct management of application and Watson-specific settings, covering hardware preferences, environment variables, and error handling.

See full tests results and explanations in **Appendix G.1**.

9.1.2 Integration API Testing

During development, the majority of the backend functionality was produced first, meaning heavy use of applications such as *Postman*. This meant testing the API endpoints continuously. As a result, test files running full sets of API endpoints were created to quickly test the system using *Pytest*. From Figure 5.1, we can see that the API endpoints sit in the respective router classes, of which there are five. This means we can create five test files respectively:

- SettingsRouter: test_settings_routes.py
- DataRouter: test_data_routes.py
- ModelRouter: test_model_routes.py
- LibraryRouter: test_library_routes.py
- PlaygroundRouter: test_playground_routes.py

We can write these tests in a particular order to replicate the user experience. We test settings first in order to authenticate models, then we move on to testing data routes. We test these first for a very important reason; when it comes to testing the model endpoints, we can configure models and test the RAG system by inferencing twice between a model configuration save to compare the models' response. We can then test the playground routes, although ordering for this test does not particularly matter as long as it occurs after the settings test.

See full tests results and explanations in **Appendix G.2**.

9.2 Error & Exception Handling

Error testing was vitally important for IBM cloud authentication used within the *download* and *load* methods in **WatsonModel** and **WatsonService**. Testing was used to force particular errors, for example if the user deletes their API key after downloading a model, and then tries to load it.

Model interactive functions mainly throws the *ModelError* exception from a set of defined exception classes, which is used when there is a direct issue concerning model interaction during download, load, and inference stages.

9.3 Model Testing & Limitations

As seen in earlier chapters, the AI models need to be tested themselves, in particular the Watsonx AI Foundation Models. In section 6.3.2.1, the response is not good enough to be used within an AI application, which leads us to use more advanced inferencing, seen in section 6.3.2.2.

Research was conducted into setting optimal prompts and parameters for each model to serve as default model configurations; in which the user may restore configurations after changing them if they wished. However, it was proved to be difficult to test this aspect due to token limit constraints for watson services using student accounts with feature codes.

9.4 User Acceptance Testing

User Acceptance Testing (UAT) involved a range of users testing the application directly, as well as reviewing demonstration videos of the app in use. We have already seen some UAT in Section 6.4.

9.4.1 Presentation Feedback

The AI Islands app was presented to stakeholders and IBM officials after the development phase was completed, in which positive feedback was received. The presentation involved showing a video demonstration of the app being used for a specific use case called *The Audio Image Description System*, as described in section 7.2.3.

9.4.2 Practical User Acceptance Test

The users who tested the app directly were encouraged to read the IBM Cloud Guide and the AI Islands User Manual in Appendix C & H respectively, before reviewing the AI Islands application. All users were of differing technical knowledge, highlighting the ease of access solution from 1.3.

9.4.2.1 Test Process

The UAT aimed to verify major aspects of the application, particularly the integration with IBM services. It focused on assessing the customisation available for creating AI model chaining systems, as well as the usability of the interface for configuring models and inferencing models. Additionally, the UAT evaluated the management of datasets through the RAG system, ensuring effective use of these datasets for LLM context. For the test case details and the user feedback, please see Table G.1 in **Appendix G.3**.

9.4.2.2 Conclusion

According to the test cases, AI Islands successfully integrates and customises AI models with IBM services, offering extra functionality such as RAG. While the application demonstrates strengths in model choice, inferencing, and customisation, improvements in UI guidance and playground model chaining instructions will enhance the user experience.

9.5 Chapter Summary

This chapter provided an overview of the various testing methods employed during the development of AI Islands, including unit testing, integration API testing, error handling, and model testing. The focus was on ensuring that key functionalities, particularly the integration with IBM services, were thoroughly tested. Error and exception handling were critical in validating cloud authentication and model interaction. Furthermore, user acceptance testing (UAT) confirmed that the app is functional, customisable, and user-friendly, though some areas, such as UI guidance and model chaining workflows, could benefit from further improvement. Overall, the testing phase has validated the application's core objectives, ensuring a solid foundation for further refinement.

Chapter 10

Conclusion

In this section, the AI Islands project is evaluated, highlighting its success in integrating multiple AI models within a single application. We begin by outlining the core goals met, while performing a critical evaluation. We will then discuss challenges and limitations encountered, and finish with a future work section, identifying potential areas for expansion.

10.1 Summary of Achievements

The achievement table in Table 10.1 shows the completion of each requirement and AI Islands as a whole. Project participation is marked using the following scale.

✓✓= Project Exclusive, ✓= Major Involvement, (✓) = Supporting, × = No Involvement, - = Pass

ID	Priority	State	Participation
Functional Requirements			
FR-1	Must have	Completed	✓
FR-2	Must have	Completed	✓
FR-3	Must have	Completed	✓
FR-4	Must have	Completed	✓
FR-5	Must have	Completed	✓
FR-6	Must have	Completed	(✓)
FR-7	Must have	Completed	✓
FR-8	Must have	Completed	✓✓
FR-9	Should have	Completed	✓✓
FR-10	Should have	Completed	✓✓
FR-11	Should have	Completed	(✓)
FR-12	Should have	Completed	(✓)
FR-13	Should have	Completed	✓✓
FR-14	Should have	Completed	✓✓
FR-15	Should have	Completed	✓
FR-16	Could have	Completed	×
FR-17	Could have	Completed	✓
FR-18	Could have	Completed	(✓)
FR-19	Could have	Completed	✓✓
FR-20	Could have	Completed	✓✓
FR-21	Could have	Completed	✓
FR-22	Won't have	×	-

Functional Requirements			
NFR-1	Must have	Completed	✓
NFR-2	Must have	Completed	✓
NFR-3	Should have	Completed	✓
NFR-4	Should have	Completed	✓
NFR-5	Should have	Completed	✓
NFR-6	Should have	Completed	✓
NFR-7	Should have	Completed	✓
NFR-8	Should have	Completed	✓✓
NFR-9	Should have	Completed	✓
NFR-10	Could have	Partially Completed	✓
NFR-11	Could have	Partially Completed	(✓)
Key Functionalities (Must have and Should have)		100% Completed	
Optional Functionalities (Could have)		87% Completed	

Table 10.1: Table of Project Achievements

10.2 Critical Evaluation

The application AI Islands addresses the direct needs of the client to be a comprehensive AI model management and inference (C# with .NET MAUI) platform. The system allows users to interact with various AI models, configure them, and run inferences across different domains such as computer vision, natural language processing, and text generation. Strong features such as model chaining and retrieval-augmented generation have enhanced the final product and expanded the user base. This application involved the implementation of a wide variety of complex features and this was executed well. In general, it has been completed successfully and is well-received by users and stakeholders.

10.2.1 User Interface Design and User Experience

The design process adhered to Nielsen's principles of user interface design. Throughout the development phase, the design was iteratively refined to incorporate feedback from both users and clients, ensuring an improved and user-centered experience. The dynamic user interface generates UI components based on selected AI models, along with detailed model configurations for each model within the index. Reports and information is available to the user through the UI, making a better user-informed experience. The general minimalist coherent design provides enhanced user learnability.

10.2.2 Functionality

All *Must Have* and *Should Have* requirements outlined in the *MoSCoW* table have been successfully fulfilled, along with the majority of the *Could Have* features, resulting in a comprehensive application. In addition to meeting these core objectives, several additional functionalities were implemented, demonstrating foresight and a commitment to enhancing the app's scalability and adaptability. These enhancements not only address current project goals but also position the application for future growth, offering flexibility for further integration and feature expansion.

10.2.3 Stability, Efficiency and Compatibility

AI Islands, built as a desktop application using .NET MAUI with a Python backend powered by FastAPI, was developed with stability in mind to ensure smooth operation across various AI models. The integration of watsonx models, alongside other locally deployed models, is central to the app's functionality. Although the app has undergone basic functionality testing, future plans involve more comprehensive testing to fully validate its ability to handle more complex model operations.

AI Islands leverages the efficiency of IBM's watsonx cloud-based models alongside offline models, creating a flexible environment where users can choose the most appropriate models for their tasks. The backend ensures that requests between the desktop front-end and the model-processing backend are handled quickly. Additionally, the app's model management system allows users to configure and chain models together, optimising inferencing workflows. The flexibility in switching between offline and online models, due to external factors, can minimise processing delays.

AI Islands was designed for cross-platform compatibility, ensuring that users on different operating systems such as Windows, macOS, and Linux can use the application. Its MAUI framework ensures that the desktop application works seamlessly across these platforms, while the backend ensures compatibility with Python-based AI models, including those from watsonx and other popular frameworks such as Huggingface Transformers and Ultralytics. The UI is responsive and intuitive, allowing users to easily navigate and access key functionalities regardless of the platform, ensuring accessibility for a wide range of users.

10.2.4 Maintainability

The architecture of AI Islands makes it highly maintainable, allowing for straightforward updates and additions to both the frontend and backend. Given the app's primary goal of integrating various models, including watsonx, into a cohesive environment, each model or feature can be managed independently, reducing the complexity of future updates.

The backend, with its clear structure, ensures that additional models or features can be integrated with minimal disruption to existing functionality. The app is structured in a way that allows for ongoing development and scaling as new AI models and user requirements emerge. Future updates, including extended testing and optimisation, can be implemented efficiently, ensuring the app continues to meet evolving demands.

10.3 Challenges

Due to the nature of this project using IBM Cloud AI models, this project was executed with student cloud accounts which do not provide many credits for model inferencing. This created a huge problem for testing the model accuracy and attempting to enhance through the use of the RAG system.

To fully integrate watsonx into an AI *Island* themed desktop application, it is recommended that the developer's IBM Cloud account has **access to unlimited tokens**. This allows for unrestricted testing, enabling the developer to explore chatbot and RAG enhancements, rigorously test with large token inputs, and evaluate model performance by adjusting parameters and prompts.

Additionally, it is advised that a **paid account** is used within the next iteration of this project. This would allow for more IBM services to be included within AI Islands, and for cost analysis features to be experimented with.

10.4 Future work

Looking ahead, there are several key areas where the AI Islands application can be further developed. One major focus will be on enhanced error handling, where more robust mechanisms will be implemented to improve user feedback and provide clearer guidance in case of issues. This will contribute to a smoother user experience, particularly when dealing with complex AI operations.

Further development will also focus on expanding the playground functionality, allowing for more sophisticated model chaining and experimentation. This enhancement will enable users to create complex workflows and explore more advanced AI scenarios. The addition of more model types should also be considered, especially in the computer vision category.

Furthermore, the addition of live inferencing would enhance the application even more. This would include using a live speech to text system, which is particularly useful as an AI tool.

The costing problem for online models could potentially be addressed. As seen in the introduction to the IBM CLI, it is possible to obtain account billing information programmatically. The developer should look into providing this information through the app's UI, and using cost analysis to alter the inference completion or output quality.

The options for online cloud-based run models could expand hugely. IBM have recently introduced a *bring your own model* (BYOM) feature to the cloud. This would allow a user to run uploaded fine-tuned models on their cloud account; the developer could look into programmatically uploading and inferencing models from the AI Islands interface.

Finally, the RAG system availability could be expanded drastically to include the option for offline LLMs. The RAG system itself could also be developed so that multiple datasets can be used within the same context prompt, allowing for more use cases to develop and higher model response accuracy.

10.5 Project Conclusion

AI Islands represents a significant step towards promoting access to advanced AI models from multiple sources. By providing a user-friendly interface for interacting with complex AI models, it enables both technical and non-technical users to utilise AI across various domains. The application's flexibility in supporting different model types position it well for future advancements in AI technology. As new models and techniques emerge, they can be readily integrated into the platform. Moreover, the inclusion of features like the playground, detailed configuration options, and RAG, promotes experimentation and learning, potentially accelerating AI adoption and innovation across industries.

In conclusion, this project demonstrates the potential of creating accessible, versatile, and powerful AI tools that bridge the gap between cutting-edge AI research and practical, real-world applications. Its impact extends beyond mere technical achievement, potentially influencing how AI is perceived, utilised, and integrated into various sectors of society and industry.

10.6 AI Islands Integration With Watsonx Demonstration Video

As part of this project, a demonstration video was produced to accompany this report. The video can be found within the supplementary materials, or alternatively by clicking this link:

`https://www.youtube.com/watch?v=7k6C9d_55YY`

References

[1] IBM. IBM Cloud Object Storage.
https://www.ibm.com/products/cloud-object-storage.

[2] IBM. IBM Kubernetes Service.
https://www.ibm.com/products/kubernetes-service.

[3] IBM. IBM watsonx Assistant.
https://www.ibm.com/products/watsonx-assistant.

[4] IBM. IBM Maximo.
https://www.ibm.com/products/maximo.

[5] IBM. IBM watsonx.ai Foundation Models.
https://www.ibm.com/products/watsonx-ai/foundation-models.

[6] IBM. IBM watsonx.ai.
https://www.ibm.com/products/watsonx-ai.

[7] IBM. IBM watsonx.data.
https://www.ibm.com/products/watsonx-data.

[8] IBM. IBM watsonx.governance.
https://www.ibm.com/products/watsonx-governance.

[9] IBM. IBM Watson Studio.
https://www.ibm.com/products/watson-studio.

[10] IBM. IBM Cloud Pak for Data.
https://www.ibm.com/products/cloud-pak-for-data.

[11] IBM. watsonx.ai.
https://www.ibm.com/products/watsonx-ai.

[12] IBM. Feature differences between watsonx deployments.
https://www.ibm.com/docs/en/watsonx/saas?topic=watsonx-feature-differences-between-deployments.

[13] G2. Compare IBM Cloud Pak for Data and IBM watsonx.ai.
https://www.g2.com/compare/ibm-cloud-pak-for-data-vs-ibm-watsonx-ai.

[14] IBM. Comparison of IBM watsonx as a Service and Cloud Pak for Data as a Service.
https://www.ibm.com/docs/en/watsonx/saas?topic=watsonx-comparison-cloud-pak-data-as-service.

[15] IBM. Credentials for Programmatic Access.
https://dataplatform.cloud.ibm.com/docs/content/wsj/analyze-data/fm-credentials.html?context=wx&audience=wdp.

[16] IBM. Tuning Studio.
https://dataplatform.cloud.ibm.com/docs/content/wsj/analyze-data/fm-tuning-studio.html?context=wx&audience=wdp.

[17] IBM. Watson Discovery.
https://www.ibm.com/products/watson-discovery.

[18] Learn Prompting. OpenAI Playground.
https://learnprompting.org/docs/basics/openai_playground.

[19] OpenAI. OpenAI API Reference Documentation.
https://platform.openai.com/docs/api-reference/introduction.

[20] LangChain. LangChain Website.
https://www.langchain.com/langchain.

[21] Jakob Nielsen. 10 usability heuristics for user interface design.
https://www.nngroup.com/articles/ten-usability-heuristics.

[22] Expert App Devs. Top 10 Reasons to Choose Python for Artificial Intelligence Project Backend Development.
https://www.expertappdevs.com/blog/choose-python-for-ai-project#:~:text=Python%20is%20easy%20to%20learn,robust%20web%20or%20mobile%20app.

[23] IBM. Python library.
https://dataplatform.cloud.ibm.com/docs/content/wsj/analyze-data/fm-python-lib.html?context=wx.

[24] Springboard. Which is Better for AI: Java or Python?
https://www.springboard.com/blog/data-science/which-is-better-for-ai-java-or-python/.

[25] Electron. Electron.
https://www.electronjs.org/.

[26] IBM. Managing user API keys.
https://cloud.ibm.com/docs/account?topic=account-userapikey&interface=ui#create_user_key.

[27] IBM. Natural Language Understanding - Authentication - Access between services.
https://cloud.ibm.com/apidocs/natural-language-understanding?code=python#granting-access.

[28] IBM. Generating an IBM Cloud IAM token by using an API key.
https://cloud.ibm.com/docs/account?topic=account-iamtoken_from_apikey&code=python#iamtoken_from_apikey.

[29] IBM. Inferencing a Foundation Model Programmatically.
https://dataplatform.cloud.ibm.com/docs/content/wsj/analyze-data/fm-prompt-notebook.html?context=wx&audience=wdp.

[30] IBM. Natural Language Understanding - Analyse.
https://cloud.ibm.com/apidocs/natural-language-understanding?code=python#analyze.

[31] Medium. Understanding Prompting, Prompt Engineering and In-Context Learning in LLMs.
https://medium.com/codex/understanding-prompting-prompt-engineering-and-in-context-learning-in-llms-2b59fb398fef.

[32] Swim. LLM Context Windows: Basics, Examples, and Prompting Best Practices.
https://swimm.io/learn/large-language-models/llm-context-windows-basics-examples-and-prompting-best-practices.

[33] IBM. Techniques for overcoming context length limitations.
https://dataplatform.cloud.ibm.com/docs/content/wsj/analyze-data/fm-context-length.html?context=wx.

[34] Pinecone. Less is More: Why Use Retrieval Instead of Larger Context Windows.
https://www.pinecone.io/blog/why-use-retrieval-instead-of-larger-context/.

[35] IBM. Sending fewer tokens with RAG inputs.
https://dataplatform.cloud.ibm.com/docs/content/wsj/analyze-data/fm-context-length.html?context=wx#rag.

Appendix A

Project Management

A.1 Project Structure

The project structure is defined in three phases; Research and Design, Development, Testing and Reporting. Phase one consists of requirement gathering, stakeholder meetings and UI designing. Phase two consists of the main application development. Phase three consists of testing the application and reporting project findings. Please see the Gantt Chart below in **Figure A.1**.

	TASK DETAIL	RESEARCH & DESIGN				PROJECT DEVELOPMENT								REPORTING		
	DATE	03/06 - 01/07				01/07 - 02/09								02/09 - 23/09		
		WEEK 1 - 4				WEEK 5 - 13								WEEK 14 - 16		
	TASK TITLE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	**Requirement Analysis**																
1.1	Project Brief Understanding	■															
1.2	Requirement Analysis		■	■													
1.3	Stakeholder Meetings	■	■			■	■										
2	**Research**																
2.1	Main Research	■	■	■	■	■	■										
2.2	IBM Cloud Platform		■	■	■												
2.3	Watsonx AI Tools			■	■												
2.4	Experimental App Prototypes			■	■	■	■										
3	**UI Design**																
3.1	Similar App Reviews				■												
3.2	Figma Prototype				■												
4	**Backend Development**																
4.1	Main Backend					■	■	■	■	■	■						
4.2	RAG System Backend							■	■								
5	**Frontend Development**																
5.1	Main Frontend									■	■	■	■	■			
5.2	RAG System Frontend										■						
6	**Testing & Reporting**																
6.1	Backend Testing								■						■	■	
6.1	Frontend Testing														■	■	■
6.3	Model Performance Testing								■					■	■		
	Project Reports														■	■	■
7	**Deployment & Handover**																
7.1	App Packaging																■
7.2	Project Handover																■
8	**Presentations & Preperation**																
8.1	IBM Headquarters				■												
8.2	UCL Presentations														■		

Figure A.1: AI Islands Gantt Chart

Appendix B

Use Case Specifications

B.1 Model Index Specifications

Name:	ViewModelIndex
ID:	UC1
Brief Description	Users can view the AI model index.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	None.
Main Flow:	1. The use case starts when a user navigates to the AI index. 2. Users can scroll through the AI index to view.
Post-conditions:	None.
Alternative Flows:	None.

Name:	DownloadModel
ID:	UC2
Brief Description	Users download models through physical downloads or authentication processes.
Primary Actor(s):	User
Secondary Actor(s):	IBM Cloud
Pre-conditions:	For online models, users must have specific resources added to their IBM Cloud resource list and a valid API key saved into the AI Islands. For offline models, users must have the necessary computer specifications to meet the download requirement.
Main Flow:	1. The use case starts when the user navigates to the model index. 2. The actor can download models by clicking the add to library button. 3. A pop-up terminal will open to show the download progress. 4. On completion, the terminal will automatically close. 5. A download success message is shown to the user.
Post-conditions:	Downloaded model is added to the users' library index.
Alternative Flows:	Download fails due to the above pre-conditions not being met. An error message with feedback is shown to the user, and no download occurs.

B.2 Model Index & Library Specifications

Name:	SearchAndFilter
ID:	UC3
Brief Description	Users can search and filter index pages.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	User is on an index page.
Main Flow:	1. Users can search using the search bar or filter by clicking the filter button, producing the filter pop-up. 2. Users can make their filter selections, submit, or reset. 3. The index updates with filtered results.
Post-conditions:	None.
Alternative Flows:	None.

B.3 Library Specifications

Name:	ViewLibrary
ID:	UC4
Brief Description	Users can view their library index.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	None.
Main Flow:	1. User navigates to the library, and can scroll through index.
Post-conditions:	None.
Alternative Flows:	None.

Name:	Load/Unload Model
ID:	UC5
Brief Description	Users can load or unload models into child processes, ready for inferencing.
Primary Actor(s):	User
Secondary Actor(s):	IBM Cloud.
Pre-conditions:	For online models, users must have the same pre-condition as UC2 with the addition of a valid project ID for Watsonx AI models. For offline models, users must have the necessary computer specifications to meet the loading requirement.
Main Flow:	1. The use case starts when a user navigates to the library. 2. Users can load / unload models here.
Post-conditions:	Child processes are created or destroyed.
Alternative Flows:	Load fails due to the above pre-conditions not being met. An error message with feedback is shown to the user, and no load occurs.

Name:	ViewModelDetail
ID:	UC6
Brief Description	Users can read information about models and check their live hardware usage.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	For the case of hardware usage, this is only shown if the model has been loaded.
Main Flow:	1. Users click on a model within their library. 2. Users are directed to the model information view.
Post-conditions:	None.
Alternative Flows:	None.

Name:	InferenceModel
ID:	UC7
Brief Description	Users can inference models by prompting them with inputs, which produces a response.
Primary Actor(s):	User
Secondary Actor(s):	IBM Cloud.
Pre-conditions:	To inference a model, the user must have successfully loaded the model.
Main Flow:	1. Users click on a model within their library. 2. Users navigate to the inference tab. 3. Users enter a prompt and submit. 4. Users view the model response.
Post-conditions:	None.
Alternative Flows:	Inference fails and produces an error due to user account token limits; either too many token inputs or they have reached their maximum spend.

Name:	FineTuneModel
ID:	UC8
Brief Description	Users can fine tune models with a dataset and parameter selection choice.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	None.
Main Flow:	1. Users click on a model within their library. 2. Users navigate to the fine-tune tab. 3. Users upload a dataset and select parameters. 4. Users execute the process. 5. A fine-tuned model is added to the users' library.
Post-conditions:	A fine-tuned model is added to the application data and the users' library.
Alternative Flows:	None.

Name:	ConfigureModel
ID:	UC9
Brief Description	Users can view model configurations, check their values, and read the optimised values. They can edit before taking action on how to configure them.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	None.
Main Flow:	1. Users click on a model within their library. 2. Users navigate to the configure tab. 3. Users can expand certain configuration sections and read about the optimised values. 4. Users can edit configurations or reset them without changing the system database.
Post-conditions:	None.
Alternative Flows:	None.

Name:	SaveAsNew
ID:	UC10
Brief Description	Users can save a model configuration to a new model under a chosen ID.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	None.
Main Flow:	Following from UC9: 1. Users click the save as new model button. 2. Configurations currently set are saved to the database under a new model ID. 3. Success message is shown. 4. Message box pops up asking the user if they wish to remain on the page or be re-directed to the library to view the new model. 5. If chose to remain, the configurations automatically refresh to show the previous models' configuration. If chose to re-direct, the user is sent to their library index.
Post-conditions:	None.
Alternative Flows:	Save as new fails due to user entering an identical ID to one that already exists in the database.

Name:	SaveModelConfig
ID:	UC10
Brief Description	Users can save a model configuration to the current model being viewed.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	None.
Main Flow:	Following from UC9: 1. Users click the save button. 2. Configurations currently set are saved to the database. 3. Success message is shown.
Post-conditions:	None.
Alternative Flows:	None.

Name:	RestoreDefaults
ID:	UC12
Brief Description	Users can restore a model configuration defined in the model index database to the current model being viewed.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	None.
Main Flow:	Following from UC9: 1. Users click the restore default button. 2. Model configurations are restored from the database. 3. Confirmation is asked for abort or restore. 4. A success message is shown in the case of restore. 5. Page updates to show the restored configurations.
Post-conditions:	None.
Alternative Flows:	None.

B.4 API Access Specifications

Name:	APIAccess
ID:	UC13a
Brief Description	Users can view API access information to interact with their downloaded models via external applications.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	The model must be present in the users' library.
Main Flow:	1. Users click on a model within their library. 2. Users navigate to the API access tab. 3. Users can copy API endpoints and use them elsewhere.
Post-conditions:	None.
Alternative Flows:	None.

Name:	APIAccess
ID:	UC13b
Brief Description	Users can view API access information to interact with their playgrounds via external applications.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	The playground exists.
Main Flow:	1. Users click on a playground within their playgrounds. 2. Users navigate to the playground API access tab. 3. Users can copy API endpoints and use them elsewhere.
Post-conditions:	None.
Alternative Flows:	None.

B.5 Playground Specifications

Name:	ViewPlaygrounds
ID:	UC14
Brief Description	Users can view their created playgrounds.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	None.
Main Flow:	1. Users navigate to the playgrounds page using the menu. 2. Users can scroll through their playground index and read their descriptions.
Post-conditions:	None.
Alternative Flows:	None.

Name:	CreatePlayground
ID:	UC15
Brief Description	Users can create a new playground with their chosen name and description, which are later editable.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	None.
Main Flow:	1. Users navigate to the playgrounds page using the menu. 2. Users can click create playground, producing a pop-up. 3. Users enter the name and description (optional), and save. 4. The UI updates automatically to show the new created playground.
Post-conditions:	None.
Alternative Flows:	None.

Name:	ManageModels
ID:	UC16
Brief Description	Users can manage their downloaded models within a playground. They can simply add or remove them here.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	Playground model chain is not loaded.
Main Flow:	1. Users navigate to a playgrounds' model manager page. 2. Users can click an add models button, producing a windowed view of the users' library. 3. Users can add models in the windowed view. 4. Users can remove models by clicking the models' respective bin icon.
Post-conditions:	None.
Alternative Flows:	None.

Name:	ConfigureChain
ID:	UC17
Brief Description	Users can change the chain order of models by using a ordered list of drop-down selectors populated with the playgrounds' models.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	Playground model chain is not loaded.
Main Flow:	1. Users navigate to a playgrounds' chain order page. 2. Users can add or remove models, and change their ordering. 3. Users can save the configuration, provided it meets the requirements.
Post-conditions:	None.
Alternative Flows:	Failed save can happen if the chain order consists of non-text-to-text intermediary models. Only the input of the first model, or the output of the last can be different from text type.

Name:	InferenceChain
ID:	UC18
Brief Description	Users can inference the playground chain.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	Playground model chain is loaded.
Main Flow:	1. Users navigate to a playgrounds' inference page. 2. Users add input and submit. 3. The response is shown.
Post-conditions:	None.
Alternative Flows:	None.

B.6 Data Specifications

Name:	UploadDataset
ID:	UC19
Brief Description	Users can upload a dataset.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	None.
Main Flow:	1. Users navigate to data refinery page. 2. Users click the find dataset, producing a file explorer selector. 3. The user can select a dataset here. 4. User clicks upload to AI Islands button to upload it.
Post-conditions:	Dataset directory is created within the app directory.
Alternative Flows:	None.

Name:	DatasetPreview
ID:	UC20
Brief Description	Users preview datasets within the data refinery.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	Dataset must have been uploaded.
Main Flow:	1. Users navigate to data refinery page. 2. Users select a dataset from the drop-down options. 3. The dataset is shown within the preview.
Post-conditions:	None.
Alternative Flows:	None.

Name:	ProcessDataset
ID:	UC21
Brief Description	Users can process datasets using embedding models.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	Dataset must have been uploaded, selected, and the chosen embedder selected.
Main Flow:	1. Users navigate to data refinery page. 2. Users select a dataset from the drop-down options. 3. The user selects their embedding type, and then selects the specific embedder model. 4. The user clicks process and the processing is executed.
Post-conditions:	Data files created within the dataset directory. Reports are generated and presented to the user.
Alternative Flows:	Processing can fail if the user has selected a watsonx embedder and has not saved a valid API key and a valid project ID to the application. The user is notified of this and no processing occurs.

Name:	ViewProcessingInfo
ID:	UC22
Brief Description	Users can view details of the dataset process.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	Dataset must have been processed.
Main Flow:	1. Users navigate to data refinery page. 2. Users select a dataset from the drop-down options. 3. The user clicks the view info which produces a pop-up. 4. The user can read the information about the processing within the pop-up.
Post-conditions:	None.
Alternative Flows:	None.

Name:	ViewProcessingReport
ID:	UC23
Brief Description	Users can view analysis of the dataset process.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	Dataset must have been processed.
Main Flow:	1. Users navigate to data refinery page. 2. Users select a dataset from the drop-down options. 3. The user clicks the view report which opens the browser. 4. The user can read the report.
Post-conditions:	None.
Alternative Flows:	None.

B.7 Settings Specifications

Name:	EditCloudSettings
ID:	UC24
Brief Description	Users can edit their IBM cloud settings in order to use watsonx services.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	None.
Main Flow:	1. Users navigate to the settings page. 2. Users enter their setting options and click save.
Post-conditions:	None.
Alternative Flows:	None.

Name:	EditDataSettings
ID:	UC25
Brief Description	Users can edit their data processing settings for RAG.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	None.
Main Flow:	1. Users navigate to the settings page. 2. Users enter their setting options and click save.
Post-conditions:	None.
Alternative Flows:	None.

Name:	EditHardwareSettings
ID:	UC26
Brief Description	Users can edit their hardware preference.
Primary Actor(s):	User
Secondary Actor(s):	None.
Pre-conditions:	None.
Main Flow:	1. Users navigate to the settings page. 2. Users enter their setting options and click save.
Post-conditions:	None.
Alternative Flows:	Fails updating if the user saves with *gpu* option, but cannot meet the requirements. If so, the drop-down resets to *cpu* to show the change never occurred.

B.8 IBM Cloud Specification

Name:	Authentication
ID:	UC27
Brief Description	IBM Cloud can authenticate a user API key, location, and project ID with their account specific information. The response determines the outcome of the user action.
Primary Actor(s):	IBM Cloud
Secondary Actor(s):	User.
Pre-conditions:	None.
Main Flow:	1. IBM Cloud receives a request. 2. A check is made in accordance with the user account. 3. A response is provided to AI Islands
Post-conditions:	None.
Alternative Flows:	None.

Name:	ResourceManagement
ID:	UC28
Brief Description	IBM Cloud can check resources within a user account, thus providing a response to AI Islands regarding model availability.
Primary Actor(s):	IBM Cloud
Secondary Actor(s):	User.
Pre-conditions:	None.
Main Flow:	1. IBM Cloud receives a request. 2. A check is made in accordance with the user account. 3. A response is provided to AI Islands.
Post-conditions:	None.
Alternative Flows:	None.

Name:	CloudProcessing
ID:	UC29
Brief Description	IBM Cloud runs model inferencing on their system, if a valid API call has been made to it. Once finished processing, it sends the response back.
Primary Actor(s):	IBM Cloud
Secondary Actor(s):	User.
Pre-conditions:	None.
Main Flow:	1. IBM Cloud receives a request for inferencing. 2. A check is made in accordance with the user account. 3. Processing occurs on the cloud. 4. A response is provided to AI Islands.
Post-conditions:	None.
Alternative Flows:	None.

	Use Case (UC)																												
FR	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
FR-1	×		×																										
FR-2		×																											
FR-3			×	×																									
FR-4					×																								
FR-5							×																						
FR-6														×															
FR-7																		×											
FR-8																													×
FR-9																								×					
FR-10																											×	×	
FR-11															×														
FR-12																×	×												
FR-13																			×	×									
FR-14																					×								
FR-15						×																							
FR-16								×																					
FR-17									×	×	×	×																	
FR-18													×																
FR-19																						×	×						
FR-20																									×				
FR-21																										×			
FR-22	Won't Have Requirement																												

Table B.31: Requirements Traceability Matrix

Appendix C

IBM Cloud Guide

Welcome to the IBM Cloud guide, designed to help you get started with the essential functions of IBM Cloud. This guide covers the basics, including logging into your IBM Cloud account, managing resources, and accessing key features such as Watson Studio.

We start with the login process, navigating the IBM Cloud dashboard, and handling common tasks like creating resources and managing API keys. Additionally, you'll learn how to interact with Watson Studio, including working with foundational AI models and integrating services into your projects.

With simple instructions and visual aids, this guide provides the foundational knowledge you need to begin using IBM Cloud.

C.1 IBM Cloud Access

C.1.1 Login

Content: These two pages consist of the login system. Figure C.1 shows where users can select their account email ID and Figure C.2 is where the user enters their account password.

Access: No login required.

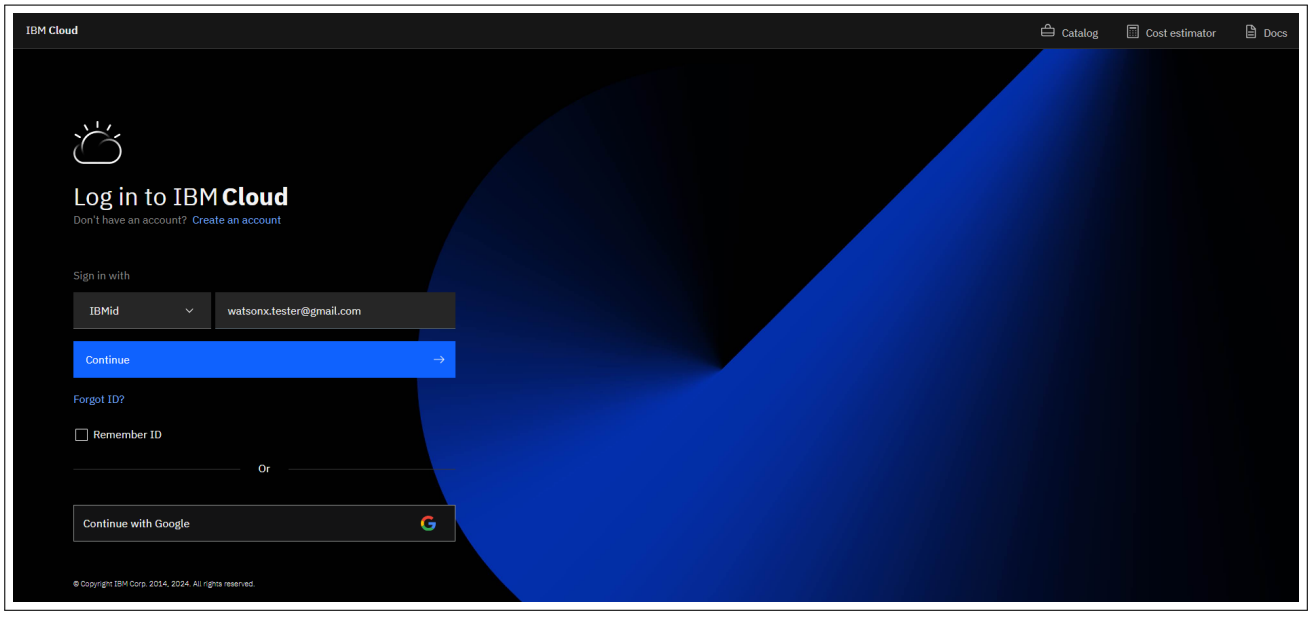


Figure C.1: IBM Cloud Login ID - Selection

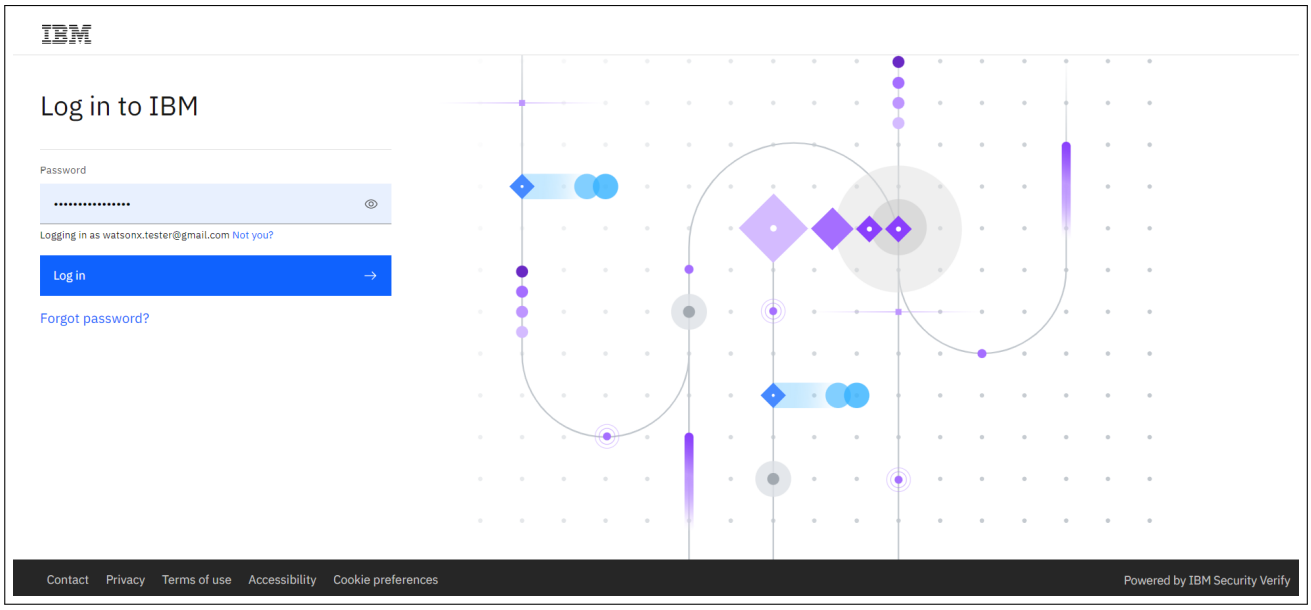


Figure C.2: IBM Cloud Login - Password

C.2 IBM Cloud Resource Management

C.2.1 Dashboard & Resource List

Content: Figure C.3 shows the user dashboard as a result of successful login and Figure C.4 is where the user can view and add a resource. They can add resources by clicking the create resource button on either page.

Access: Login required.

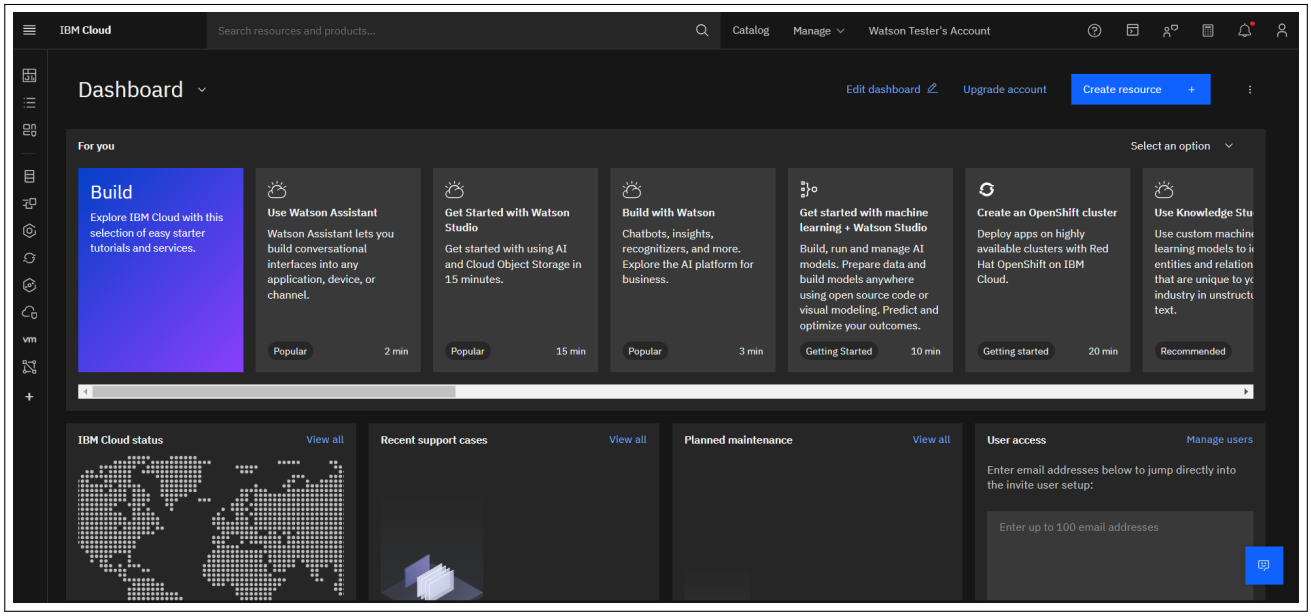


Figure C.3: IBM Cloud User Dashboard

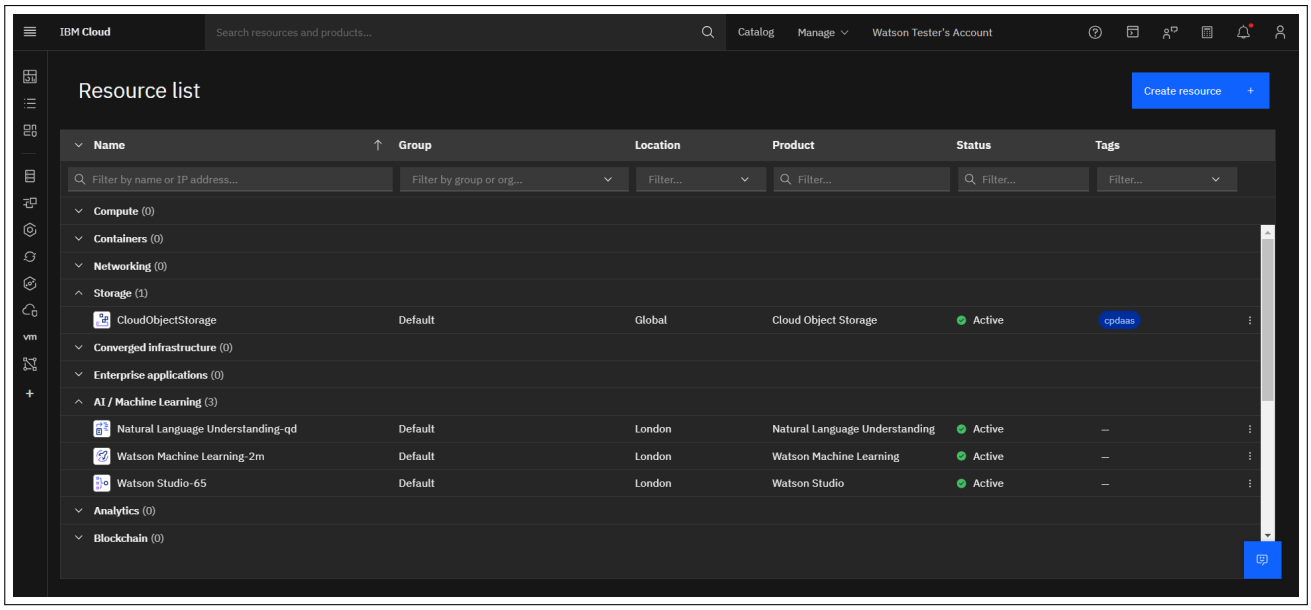


Figure C.4: IBM Cloud User Resource List

C.2.2 Resource Example: NLU

Content: Figure C.5 shows the user service page for NLU. A similar page is provided for services such as Text to Speech and Speech to Text.

Access: Login required and service resource created.

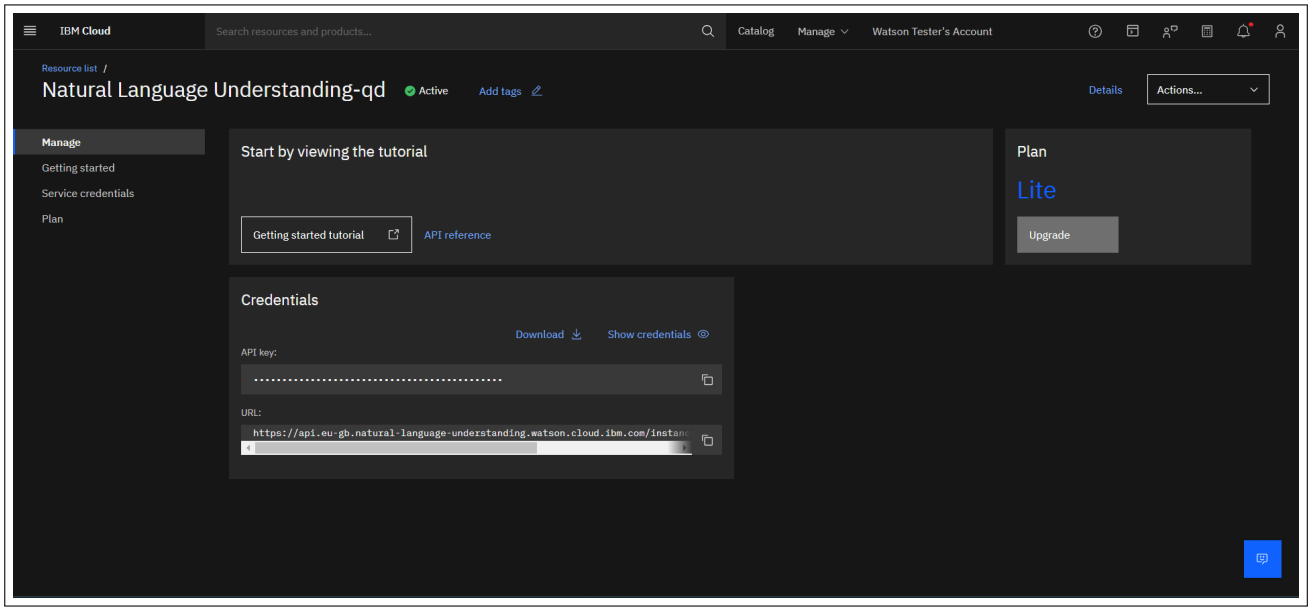


Figure C.5: IBM Cloud NLU Service Page

C.3 Access Management

C.3.1 General IAM API Key

Content: Figure C.6 shows the page for creating a general API key.

Access: Login required.

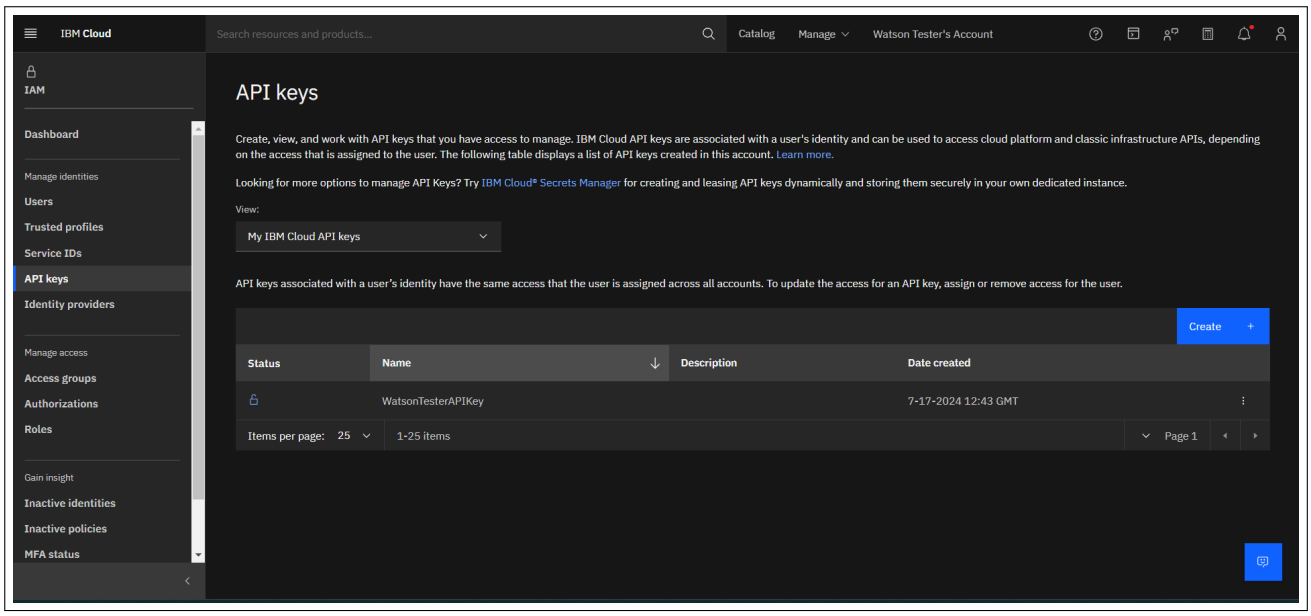


Figure C.6: IBM Cloud Access Management

C.4 Watson Studio

C.4.1 Prompt Lab

Content: Figure C.7 shows the page for creating interacting with a watsonx.ai foundation model through a chat bot and Figure C.8 shows the model index where the user can select a model to prompt.

Access: Login required and Watson Studio resource created.

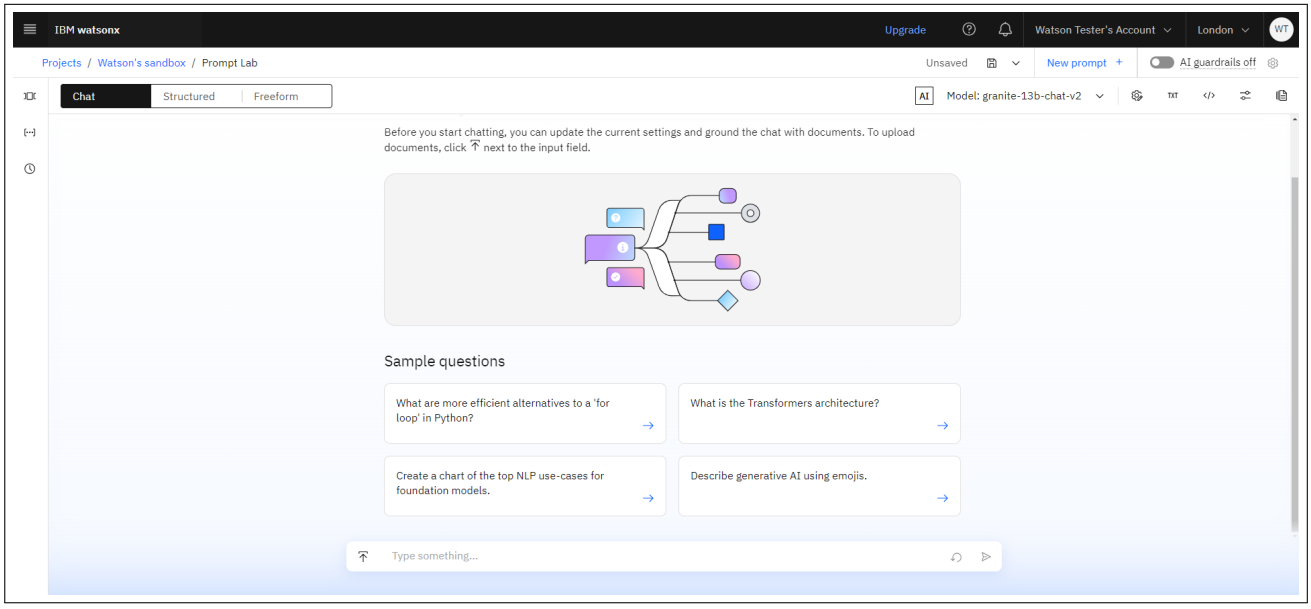


Figure C.7: IBM Cloud Watson Studio Prompt Lab

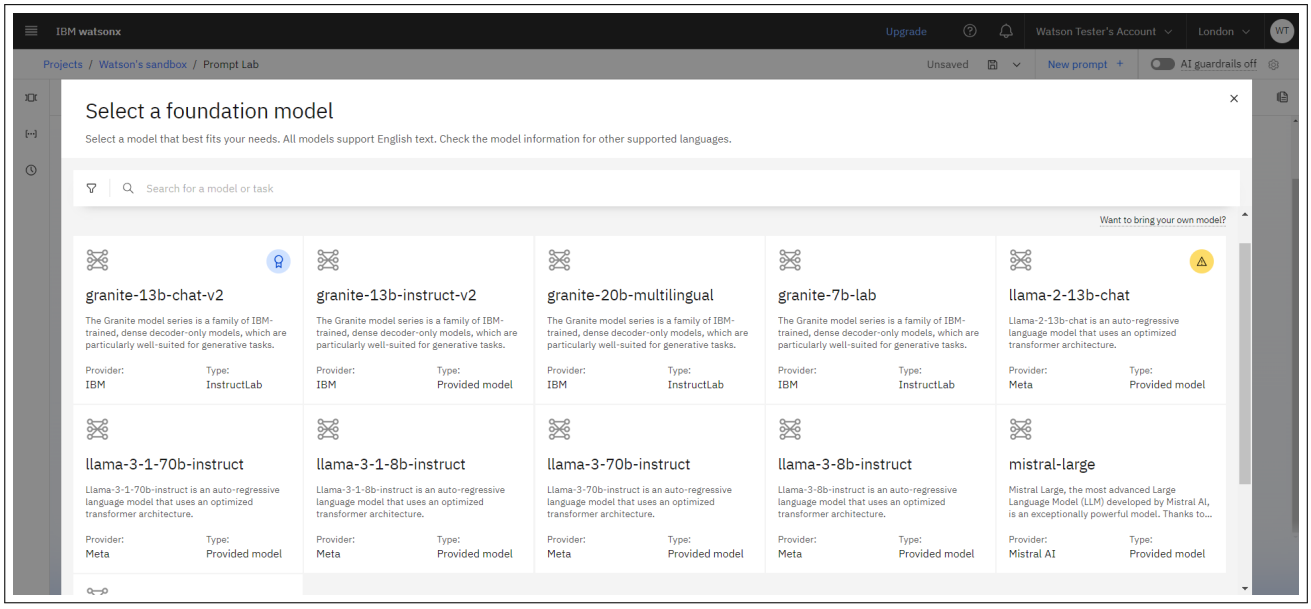


Figure C.8: IBM Cloud Watson Studio Prompt Lab Foundation Model Options

C.4.2 Watson Studio Projects

Content: Figure C.9 shows the page for viewing a users' project list and how they can create new projects. All new accounts will start off with a sandbox project by default. Figure C.10 shows how services can be integrated into a project. In order to run foundation models, the project must be associated with an instance of watson machine learning.

Access: Login required and Watson Studio resource created.

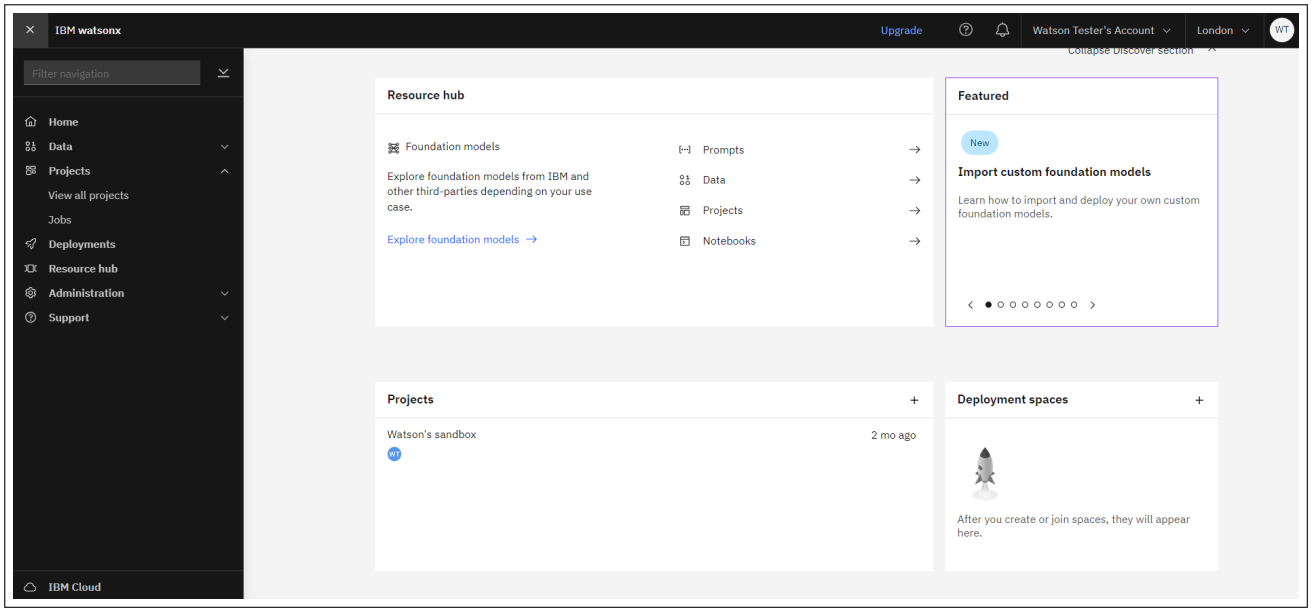


Figure C.9: IBM Cloud Watson Studio Projects

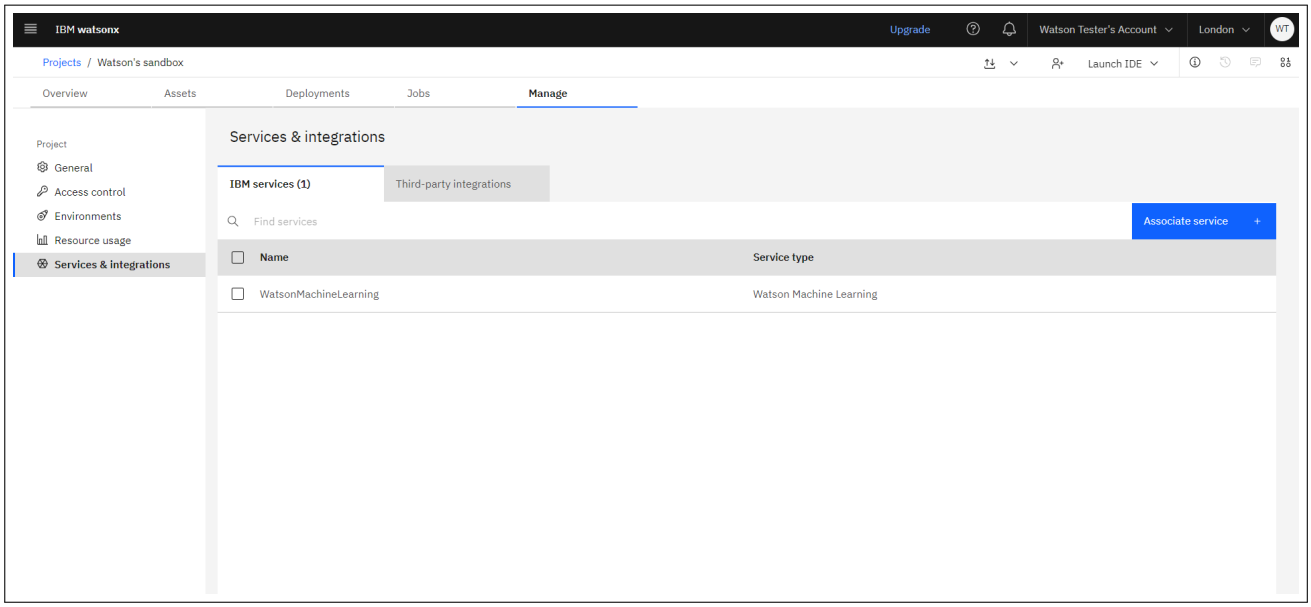


Figure C.10: IBM Cloud Watson Studio Services & Integrations

Appendix D

IBM Command Line Interface (CLI) Guide

Welcome to the IBM Command Line Interface (CLI) guide.

D.1 IBM Cloud Access

In this section, we will look at the main commands and responses seen when using the CLI.

D.1.1 CLI Login

A user can log in to the CLI using their cloud account details. Note the two methods in which *Watson Tester* was successful when using '–sso' login, as seen in Figure D.1.

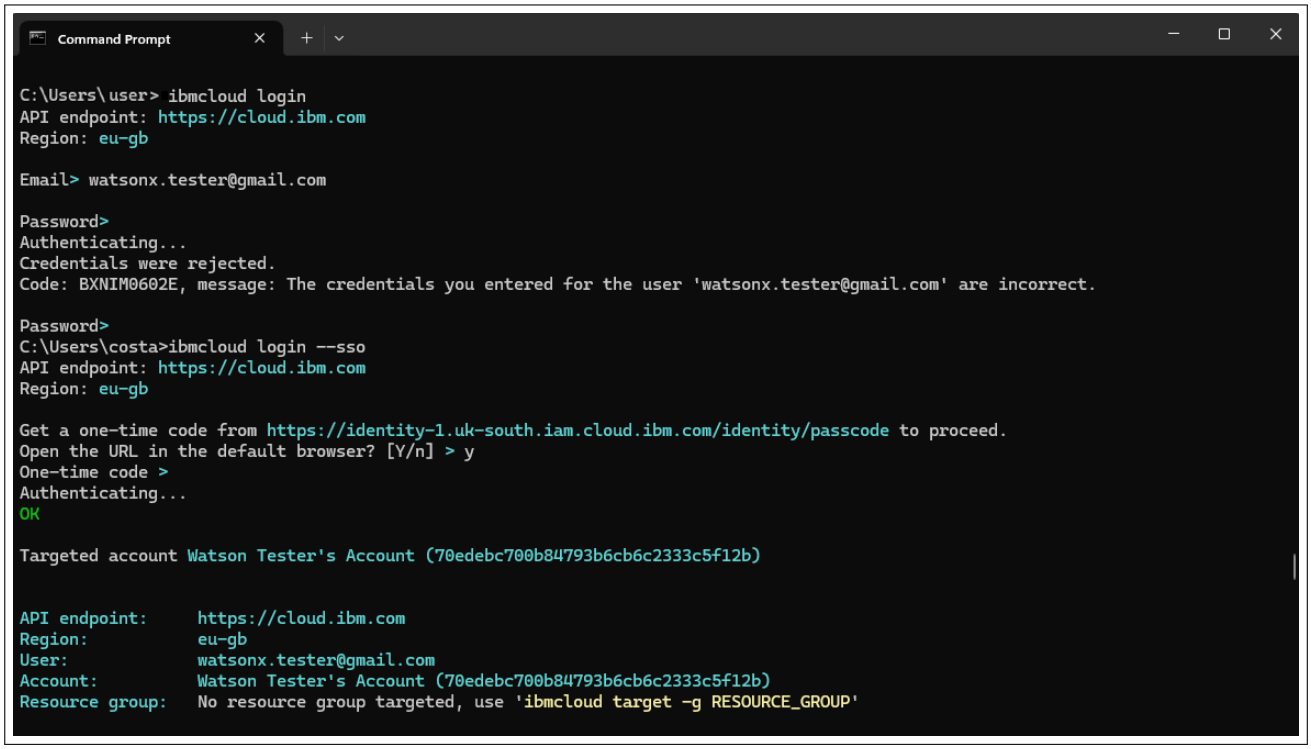


Figure D.1: IBM CLI Login

D.1.2 CLI Account Resource List

A user may want to access their current resource list. The CLI provides a useful programmatic way to check if a resource is missing, without having to check their account through the cloud website UI. Figure D.2 shows that Watson Tester can use foundation models and the natural language understanding service.

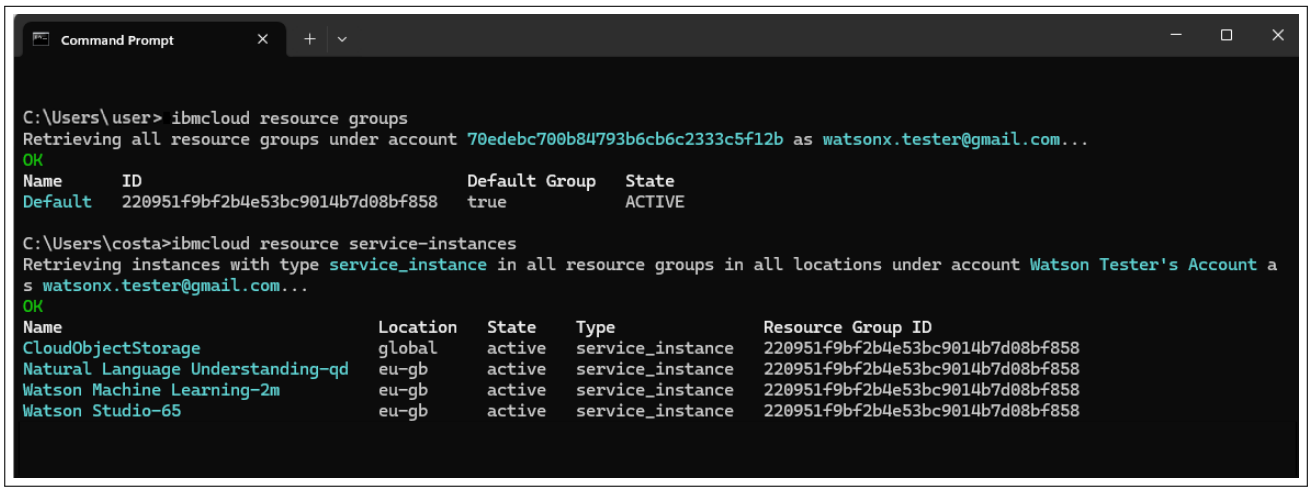


Figure D.2: IBM CLI Account Resource List

D.1.3 CLI Account Billing

A user may want to access account billing information programmatically as seen in Figure D.3. This might be useful for future development of AI Islands where costing becomes a main aspect of cloud-based run AI models.

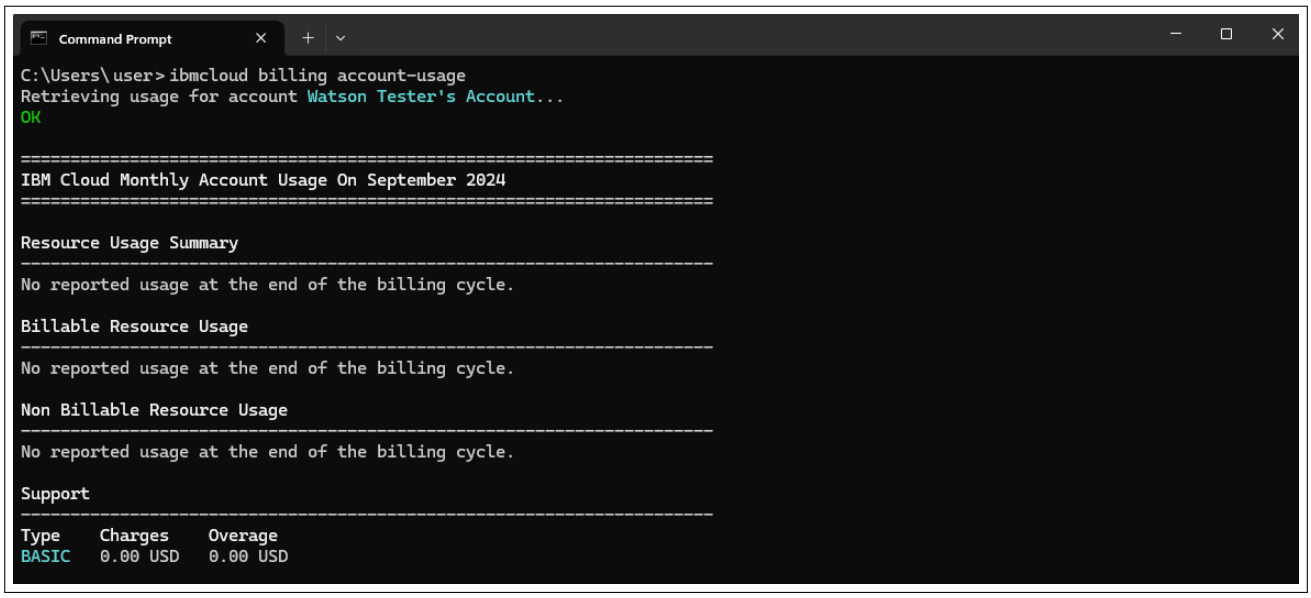


Figure D.3: IBM CLI Account Billing Information

Appendix E

Inventory

This is the inventory for AI Islands' IBM Services. Note that due to IBM being an external service, any updates can affect this inventory. As of the date of this report, the inventory is as follows.

E.1 Watsonx AI Models

The following is a full set of all models included within AI Islands from the Watsonx AI library. These models are most likely subject to change due to regular IBM service updates. However, the following have been used in AI Islands at the time of this report.

E.1.1 Foundation Models

There are 15 Foundation Models included:

- `codellama/codellama-34b-instruct-hf`
- `google/flan-t5-xl-3b`
- `google/flan-t5-xxl-11b`
- `google/flan-ul2-20b`
- `ibm/granite-13b-chat-v2`
- `ibm/granite-13b-instruct-v2`
- `ibm/granite-20b-multilingual`
- `ibm/granite-7b-lab`
- `meta-llama/llama-2-13b-chat`
- `meta-llama/llama-3-1-70b-instruct`
- `meta-llama/llama-3-1-8b-instruct`
- `meta-llama/llama-3-70b-instruct`
- `meta-llama/llama-3-8b-instruct`
- `mistralai/mistral-large`
- `mistralai/mixtral-8x7b-instruct-v01`

Each foundation model has the following configurations:

- Prompt Settings:
  - `system_prompt`: *Defines the default instructions or behavior.*
  - `example_conversation`: *Defines a template for the response.*
- Parameters:
  - `temperature`: *Controls the randomness of the response.*
  - `top_p`: *Nucleus sampling; limits the probability mass for sampling.*
  - `top_k`: *Limits the number of highest-probability tokens to sample from.*
  - `max_new_tokens`: *Maximum number of tokens that can be generated in a response.*
  - `min_new_tokens`: *Minimum number of tokens generated in a response.*
  - `repetition_penalty`: *Controls the penalty for repeating tokens.*
  - `random_seed`: Seed for reproducibility of results.
  - `stop_sequences`: *Defines sequences that will stop the generation of text.*
- RAG (Retrieval-Augmented Generation) Settings:
  - `use_dataset`: *Boolean to enable or disable the use of a dataset for RAG.*
  - `dataset_name`: *Specifies the name of the dataset to use (if any).*
  - `similarity_threshold`: *Similarity score threshold for retrieving relevant data.*
  - `use_chunking`: *Boolean for using chunked method folder.*
- Chat History:
  - `chat_history`: *Boolean for chatbot mode.*

E.1.2 Embedding Models

There are 7 Embedding Models included:

- `cross-encoder/ms-marco-minilm-l-12-v2`
- `ibm/slate-125m-english-rtrvr`
- `ibm/slate-125m-english-rtrvr-v2`
- `ibm/slate-30m-english-rtrvr`
- `ibm/slate-30m-english-rtrvr-v2`
- `intfloat/multilingual-e5-large`
- `sentence-transformers/all-minilm-l12-v2`

E.2 Watson Services

The following is a full set of all models included within AI Islands from the IBM Watson library.

E.2.1 Natural Language Understanding Service

This service is a fixed service and treated as a singular model in AI Islands with the following configuration option:

- Features:
  - `sentiment`: *Boolean on / off.*
  - `emotion`: *Boolean on / off.*
  - `keywords`: *Boolean on / off.*
  - `categories`: *Boolean on / off.*
  - `concepts`: *Boolean on / off.*
  - `relations`: *Boolean on / off.*
  - `semantic_roles`: *Boolean on / off.*

E.2.2 Text to Speech Service

This service is a fixed service and treated as a singular model in AI Islands with the following configuration option:

- Settings:
  - `voice`: *Selection of 52 voices.*
  - `pitch`: *Adjust the voice pitch.*
  - `speed`: *Adjust the voice speed.*

E.2.3 Speech to Text Service

This service is a fixed service and treated as a singular model in AI Islands with the following configuration option:

- Settings:
  - `model`: *Selection of 86 models.*
  - `content_type`: *WAV files only.*

E.3 Data Processing Models

Data processing contains the online watsonx embedding models from section E.1.1 in addition to the following offline embedders:

- `all-MiniLM-L6-v2`
- `all-mpnet-base-v2`
- `all-distilroberta-v1`
- `all-MiniLM-L12-v2`
- `paraphrase-multilingual-MiniLM-L12-v2`
- `paraphrase-multilingual-mpnet-base-v2`
- `distiluse-base-multilingual-cased-v1`
- `msmarco-distilbert-base-v4`
- `multi-qa-MiniLM-L6-cos-v1`
- `all-roberta-large-v1`
- `paraphrase-albert-small-v2`
- `paraphrase-MiniLM-L3-v2`

File types accepted for processing are *csv*, *txt*, *md*, *doc*, *docx*, and *pdf*.

E.4 Settings

The individual settings under the three categories are as follows:

- IBM Cloud Settings:
  - `api_key`: *Defines the API key for IBM Cloud account authentication.*
  - `location`: *Defines a user location to set URLs.*
  - `project_id`: *Defines a watson studio project for Watsonx AI models.*
- Global RAG Settings:
  - `use_chunking`: *Boolean for processing method.*
  - `chunk_size`: *chunk character length limit.*
  - `chunk_overlap`: *Chunk overlap.*
  - `chunk_method`: *chunk type.*
  - `rows_per_chunk`: *CSV rows per chunk.*
  - `csv_columns`: *CSV column inclusion.*
- Hardware Settings:
  - `hardware`: *Boolean for cpu or gpu.*

Appendix F

Frontend Implementation

In this section, we explore in depth the mechanics of frontend code.

F.1 Inference Implementation

F.1.1 Overview

The inference functionality has been developed to manage a variety of model types within a unified interface, ensuring flexibility for different AI tasks such as object detection, text generation, and others. This section outlines the key components, design approach, and inference flow of the system, highlighting its adaptability.

F.1.2 Key Components

The core components of the inference system are as follows:

- **Inference.xaml.cs (View)**: Handles UI logic.
- **Inference.xaml (View XAML)**: Defines the visual interface layout.
- **InferenceViewModel.cs (ViewModel)**: Manages business logic and data handling.

These components follow the Model-View-ViewModel (MVVM) pattern, ensuring clear separation between user interface (UI) logic, business logic, and data management, thereby enhancing maintainability and scalability.

F.1.3 Design Approach

F.1.3.1 Dynamic UI Generation

The system implements a dynamic UI generation strategy through the *CreateInputUI()* method in `Inference.xaml.cs`. This method generates input controls dynamically based on the model's pipeline tag, ensuring that the UI adapts to various model types without requiring individual views for each type. This design enables flexibility, supporting different models without significant alterations to the user interface.

F.1.3.2 Unified Inference Method

The *RunInference()* method in *InferenceViewModel.cs* serves as the central point for handling all inference operations. By utilising a switch statement based on the model's pipeline tag, this method determines the appropriate data format and processing logic for each model type. This unification streamlines the inference process and allows for a single entry point for all models.

F.1.3.3 Flexible Input Handling

The system is designed to support multiple input types, including text inputs and file selections. Input data is dynamically formatted to meet the requirements of the selected model before being sent for inference. The AI Islands frontend uses *MediaElement* from the .NET MAUI Community Toolkit to show video previews for the live video inferencing. This capability ensures that the system is adaptable to various use cases and model specifications.

F.1.3.4 Adaptable Output Processing

To ensure user-friendly results, the system processes output through custom formatting methods. *FormatTextClassificationOutput()* will present nicely formatted key-value pairs, where as *FormatTranslationOutput()* will preserve the original prompt structure for better readability; all character spacings are preserved which is quite useful for translator models. The *RunInference()* method contains another switch statement that processes and formats the output based on the model type, allowing for outputs to be presented in the most appropriate format for each specific task.

F.1.3.5 Support for Multiple Output Types

The system can manage different output types, including text, images, audio, and JSON data. The user interface adapts to the output format, displaying the appropriate viewer or player, such as an image viewer for object detection results or an audio player for text-to-speech outputs. The AI Islands frontend uses *MediaElement* from the .NET MAUI Community Toolkit to show audio outputs. This adaptability enhances user experience by delivering results in a format best suited to the data type.

F.1.3.6 Extensibility

The system is designed with extensibility in mind, allowing for the seamless addition of new model types. By extending the switch statements in *CreateInputUI()* and *RunInference()*, new models and output types can be incorporated with minimal changes to the existing structure. Additional formatting methods can also be introduced as required, further enhancing the system's flexibility.

F.1.4 Inference Flow

Inference follows a structured process, ensuring efficiency and adaptability across different models:

1. The user selects a model type.
2. The user interface dynamically adjusts to present the appropriate input controls.
3. The user provides the required input (e.g., text, file).
4. The *RunInference()* method is triggered, which:
    - Formats the input data based on the selected model type.
    - Sends the formatted data to the inference service.
    - Receives the results and processes them according to the model type.
    - Updates the user interface with the processed and formatted output.

This streamlined flow ensures that the system can handle a wide range of AI tasks with minimal manual intervention.

F.1.5 Conclusion

The inference system has been designed to provide a scalable, flexible, and maintainable solution for handling various AI model types within a single interface. By leveraging dynamic UI generation and a centralised inference logic, the system ensures efficient management of different models and input / output formats. This approach clears the way for easy adaptation to new models and requirements with minimal modifications to the core structure, making the system both robust and future-proof.

F.2 Configuration Implementation

F.2.1 Overview

The model configuration system is designed to provide a flexible and user-friendly interface for configuring AI models. It allows users to view and modify various settings for different types of models, with a focus on adaptability to different model configurations.

F.2.2 Key Components

The core components of the model configuration system include:

- **ModelConfig.xaml (View)**
- **ModelConfig.xaml.cs (View Code-Behind)**
- **ConfigViewModel.cs (ViewModel)**
- **Model.cs (Model)**

This structure follows the typical MVVM pattern from .NET MAUI applications, which is highly important for the case of configuration due to its complex nature.

F.2.3 Design Approach

F.2.3.1 Dynamic UI Generation (ModelConfig.xaml)

The interface uses *toolkit Expander* controls to create collapsible sections for different configuration categories, enhancing organisation and user experience, particularly useful for length configuration lists. It uses data binding to enable the display and editing of model properties directly within the UI. Additionally, converters, such as the *NullToVisibilityConverter*, are used to conditionally show or hide UI elements based on the state of model properties, ensuring a dynamic and responsive interface.

F.2.3.2 Configuration Logic (ModelConfig.xaml.cs)

The system handles user interactions and updates the ViewModel accordingly, ensuring that changes are reflected in real-time. It manages the loading and saving of model configurations to maintain consistency across sessions. Additionally, it implements methods for adding and removing items in collections, such as example conversations and candidate labels, allowing for flexible configuration management.

F.2.3.3 Data Representation (ConfigViewModel.cs)

The ConfigViewModel class acts as an intermediary between the View and the Model, allowing the flow of data and updates between them. It uses *ObservableCollection<T>* for collections such as *ExampleConversation*, *CandidateLabels*, *StopSequences*, and *LanguagesList*, ensuring that any changes to these collections are automatically reflected in the UI. Additionally, ConfigViewModel implements *INotifyPropertyChanged* through its inheritance from ObservableObject and the use of the [ObservableProperty] attribute. This setup allows properties to notify the UI of changes, ensuring that the interface remains in sync with the underlying data.

F.2.3.4 Model Structure (Model.cs)

The Model file defines the structure of the AI model, including all configurable properties necessary for its functionality. It employs attributes such as [JsonPropertyName] to ensure proper JSON serialisation and deserialisation, allowing data exchange between the application and external systems.

F.2.4 Key Features

F.2.4.1 Expandable And Collapsible Sections

The expandable and collapsible sections allow users to focus on specific configuration areas, improving the overall user experience. These sections are implemented using a toolkit expander element in View, providing a structured and organised interface for managing model configurations. Users have the option to expand all or collapse all sections for further usability.

F.2.4.2 Dynamic Property Visibility

Dynamic property visibility is managed by showing or hiding configuration options based on its existence within a models' configuration. This functionality is implemented using converters and bindings in the View, ensuring that only relevant options are displayed to the user, based on the selected model.

The main visibility converters are as follows. *NullToVisibilityConverter* ensures that UI sections are shown or hidden based on whether a property in the ViewModel is null. *AnyPropertyExistsConverter* checks whether certain configuration properties in the Config object are set, and displays relevant UI sections accordingly.

F.2.5 Configuration Types

Configurations in the application are presented to the user through intuitive UI elements, such as drop-downs and sliders, designed to enhance usability. A key component is the *OptimizedSlider* control, a custom slider created for refined experience for adjusting numerical values. This control offers several advanced features. Users can configure the range, step values, and minimum and maximum limits through bindable properties, making it versatile for different settings. The control also supports both integer and floating-point values, with customisable decimal precision for floating-point numbers. The control has been carefully designed; the developer can import the control and configure its values within the configuration View.

An additional optimised value indicator (in green) visually distinguishes the recommended or default setting, helping users make informed adjustments. Furthermore, the slider can be configured to snap to predefined fixed values, simplifying the selection of specific options. The *OptimizedSlider* is used within the configuration UI file to allow users to adjust key configuration parameters such as temperature, top-p, and top-k values, ensuring both flexibility and precision in the configuration process.

F.2.5.1 Collection Management

Collection management supports adding and removing items in collections, such as example conversations. This functionality is implemented in View Code-Behind, with methods like *OnAddExampleMessageClicked()* and *OnDeleteExampleMessageClicked()* handling these operations efficiently.

F.2.5.2 Data Validation

Data validation ensures that user inputs, such as the random seed range, are valid before being processed. This is implemented in the View Code-Behind, with methods like *OnRandomSeedTextChanged()* handling the validation to maintain data integrity and prevent errors.

F.2.5.3 Configuration Persistence

Configuration persistence allows users to save configurations and create new models based on existing ones. This functionality is implemented in the View Code-Behind through methods like *OnSaveConfigClicked()* and *OnSaveAsNewModelClicked()*, ensuring that configurations can be easily stored and reused. The frontend has been carefully designed to preserve or update configuration values based on user actions; if the user edits configuration values and saves as a new model, but chooses not to be re-directed to the library, then the configuration values will reset to the current model instead.

F.2.5.4 Reset & Restore

The reset functionality provides users with the option to discard their current edits and revert to the original configurations. This is implemented in the View Code-Behind through the *OnResetClicked()* method. When triggered, this method fetches the original model data from the library and resets the ConfigViewModel to its initial state. It ensures that all collections and properties are restored to their

original values, providing a quick way for users to discard any modifications they have made during the current session, promoting an even more user-friendly experience.

The restore defaults functionality allows users to revert completely to the default configurations defined for the model within the model index. This is implemented in the View Code-Behind through the *OnResetDefaultsClicked()* method. When invoked, this method resets the configuration settings to their predefined default values, ensuring that the model is configured according to the initial default settings. This functionality is particularly useful for users who want to start over with the default configuration without any custom modifications.

F.2.6 Data Flow

The data flow for model configuration is structured as follows:

1. Model defines the structure of the AI model configuration.
2. ViewModel wraps the Model data and provides properties and collections for the View to bind to.
3. The View displays the configuration options and binds to the ViewModel.
4. User interactions in View are handled by View Code-Behind, which updates ViewModel.
5. The converters referenced in section F.2.4.2 work in coordination with properties in ViewModel to dynamically control the visibility of UI elements based on the presence of configuration data.
6. Changes in ViewModel are shown in the UI due to data binding and property change notifications.

F.2.7 Extensibility

The extensibility of the system allows for new configuration options to be easily added by extending the Model class and updating both the ViewModel and View. The use of converters and bindings ensures that new UI elements can be seamlessly integrated to respond dynamically to changes in model properties.

F.2.8 Conclusion

This implementation provides a flexible and maintainable system for configuring AI models, while allowing extensive UI designing within the View. By separating concerns across its MVVM structure, it allows for easy updates and extensions to support new model types or configuration options. The UI provides a declarative way to create a responsive and adaptive interface for model configuration.

Appendix G

Test Results

G.1 Unit Tests

This section covers the test results for unit tests in more depth, with results provided for the reader.

G.1.1 Testing - Watson Model Functions

The file `test_watson_model_init.py` contains unit tests for the initialisation of the WatsonModel class. It verifies that the model is correctly initialised with default values for both foundation and embedding models. It also tests the *select_project()* method to ensure it properly selects and sets a project when available.

The `test_watson_model_download.py` file focuses on the download class method of WatsonModel. They verify that the method correctly handles downloading both foundation and embedding models, properly sets up the model information, and handles API key validation. It also tests error cases such as missing or invalid API keys.

The file `test_watson_model_load.py` tests the load method of WatsonModel. It verifies that both foundation and embedding models can be loaded successfully, with proper mocking of external dependencies. It also tests error cases such as invalid API keys or project IDs, and ensures that the model's attributes are correctly set after loading.

Finally, the tests in `test_watson_model_inference.py` cover the inference method of WatsonModel for both foundation and embedding models. They verify that the method correctly processes input, interacts with the underlying Watson API (mocked), and returns expected outputs. It also tests chat history functionality, error handling during API calls, and the *clear_chat_history()* method.

```
PS C:\Users\Ai-Islands> pytest backend/tests/unit/models/watson_model
============================================= test session starts =============================================
platform win32 -- Python 3.12.1, pytest-8.3.1, pluggy-1.5.0
rootdir: C:\Users\Ai-Islands
configfile: pytest.ini
plugins: anyio-4.4.0, asyncio-0.24.0, order-1.2.1
asyncio: mode=Mode.AUTO, default_loop_scope=function
collected 19 items

backend\tests\unit\models\watson_model\test_watson_model_download.py ....                    [ 21%]
backend\tests\unit\models\watson_model\test_watson_model_inference.py ......                 [ 52%]
backend\tests\unit\models\watson_model\test_watson_model_init.py ....                        [ 73%]
backend\tests\unit\models\watson_model\test_watson_model_load.py .....                       [100%]

============================================== 19 passed in 10.95s ==============================================
```

Figure G.1: Watson Model Functions Pytest

G.1.2 Testing - Watson Service Functions

The file `test_watson_service_init.py` contains unit tests for the initialisation of the WatsonService class. It verifies that the service is correctly initialised with default values for three different Watson services: Natural Language Understanding (NLU), Text-to-Speech (TTS), and Speech-to-Text (STT). The tests ensure that all attributes are properly set to their initial values upon instantiation.

The `test_watson_service_download.py` focuses on the download class method of WatsonService. They verify that the method correctly handles downloading for NLU, TTS, and STT services. The tests check successful downloads, as well as error cases such as missing or invalid API keys, and missing services in the IBM Cloud account. It uses parameterised tests to cover all three service types efficiently.

The file `test_watson_service_load.py` tests the load method of WatsonService for NLU, TTS, and STT. It verifies that each service can be loaded successfully, with proper mocking of external dependencies like API keys and service credentials. It also tests error cases such as missing API keys or service credentials. The tests use fixtures and parameterisation to cover all three service types.

Finally, the tests in `test_watson_service_inference.py` cover the inference method of WatsonService for NLU, TTS, and STT. They verify that the method correctly processes input, interacts with the underlying Watson APIs (mocked), and returns expected outputs for each service type. The tests also check error handling, such as when the model is not loaded or when API calls fail. Additionally, it includes tests for utility methods like listing available STT models.

```
PS C:\Users\Ai-Islands> pytest backend/tests/unit/models/watson_service
============================================= test session starts =============================================
platform win32 -- Python 3.12.1, pytest-8.3.1, pluggy-1.5.0
rootdir: C:\Users\Ai-Islands
configfile: pytest.ini
plugins: anyio-4.4.0, asyncio-0.24.0, order-1.2.1
asyncio: mode=Mode.AUTO, default_loop_scope=function
collected 34 items

backend\tests\unit\models\watson_service\test_watson_service_download.py ............                [ 35%]
backend\tests\unit\models\watson_service\test_watson_service_inference.py ..........                [ 64%]
backend\tests\unit\models\watson_service\test_watson_service_init.py ...                            [ 73%]
backend\tests\unit\models\watson_service\test_watson_service_load.py .........                      [100%]

============================================= 34 passed in 7.63s ==============================================
```

Figure G.2: Watson Service Functions Pytest

G.1.3 Testing - IBM Cloud Account Interaction Functions

The file `test_authentication.py` tests the Authentication class, which handles IBM Cloud API authentication. It verifies the initialisation of the class with API keys, tests the *get_iam_token()* method for both successful and failed scenarios, and checks the *validate_api_key()* and *validate_project()* methods. The tests use mocking to simulate API responses and ensure proper error handling.

The tests in `test_resource_service.py` focus on the ResourceService class, which inter-

acts with IBM Cloud resources. It includes tests for *get_resource_list()* and *get_service_credentials()* methods, covering both successful scenarios and error cases. The tests verify that the class correctly handles API responses and raises appropriate exceptions when errors occur.

The file `test_account_info.py` tests the AccountInfo class, which provides information about the IBM Cloud account. It verifies the lazy evaluation of credentials and Watson API client, tests the *list_projects()* method, and checks the *get_service_credentials()* method. The tests ensure that the class correctly interacts with other components like Authentication and ResourceService.

Finally, the tests in `test_project_functions.py` cover utility functions related to IBM Cloud projects, specifically *get_projects()* and *select_project()*. The tests verify that projects are correctly retrieved and formatted, and that project selection works as expected. It also includes error case testing, such as when token retrieval fails or when an invalid project is selected.

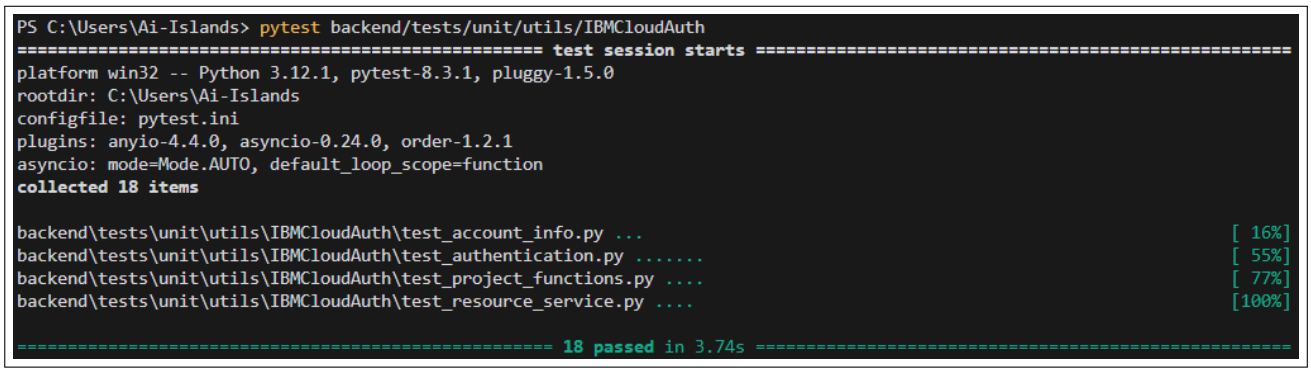


Figure G.3: IBM Cloud Account Interaction Functions Pytest

G.1.4 Testing - RAG Dataset Functions

The file `test_dataset_management.py` tests the DatasetFileManagement class, which handles file operations for datasets. It includes tests for:

- Uploading datasets
- Previewing dataset content
- Checking dataset processing status
- Deleting datasets
- Retrieving dataset processing information
- Listing available datasets
- Updating and retrieving dataset metadata
- Checking RAG settings feasibility
- Getting dataset tracker information
- Retrieving dataset reports

The tests use mocking extensively to simulate file operations and API responses, ensuring that the class correctly handles various scenarios, including error cases.

This file `test_dataset_utility.py` tests the DatasetManagement class, which handles the core functionality of dataset processing and embedding generation. Tests include:

- Getting available embedding models
- Generating embeddings using Sentence Transformers and watsonx models
- Processing datasets; reading files, generating embeddings, and creating FAISS indexes
- Finding relevant entries in processed datasets

These tests mock external dependencies like SentenceTransformer, WatsonxEmbeddings, and FAISS to isolate the functionality of the DatasetManagement class. They verify that embeddings are correctly generated, datasets are properly processed, and relevant entries can be retrieved efficiently.

Both test files ensure that the RAG utility functions work correctly, covering various aspects of dataset management, processing, and retrieval. They use *pytest* fixtures and patching to create isolated testing environments and simulate different scenarios, including success cases and error handling.

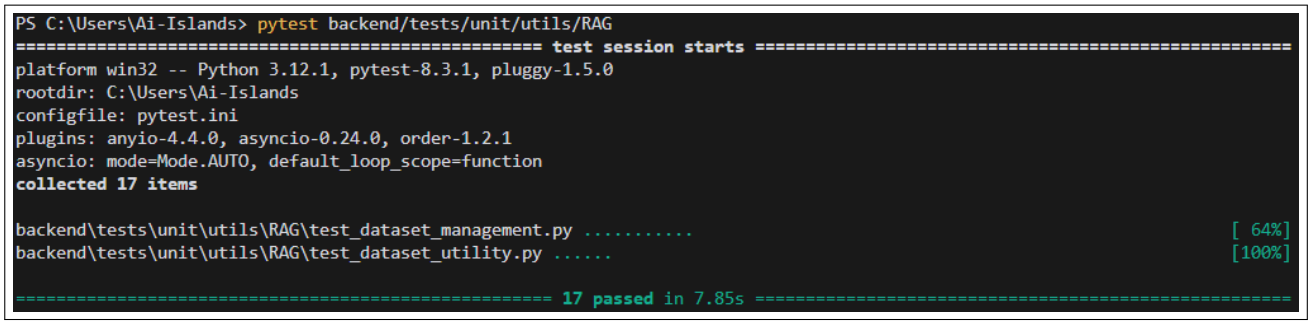


Figure G.4: RAG Dataset Functions Pytest

G.1.5 Testing - Settings Functions

The file `test_settings_service.py` tests the SettingsService class, which manages various settings for AI Islands. It includes tests for:

- Updating Watson settings
- Retrieving Watson settings
- Updating chunking settings
- Retrieving chunking settings
- Setting hardware preference (CPU/GPU)
- Getting hardware preference
- Checking GPU availability and version information

The tests use *pytest* fixtures and mocking to simulate different scenarios and isolate the SettingsService functionality from external dependencies.

In particular, `test_watson_settings_manager.py` tests the WatsonSettingsManager class, which handles IBM watson account-specific settings. Key tests include:

- Verifying single-instance behavior

- Reloading environment variables
- Getting and setting environment variables
- Updating Watson location settings
- Retrieving all Watson settings
- Creating default environment variables

These tests use mocking to simulate file operations and environment variable interactions. They ensure that the WatsonSettingsManager class correctly manages watson-specific settings and handles various scenarios, including error cases.

Both test files ensure that the settings management functions work correctly, covering various aspects of application configuration. They use *pytest* fixtures and patching to create isolated testing environments and simulate different scenarios, including success cases and error handling.

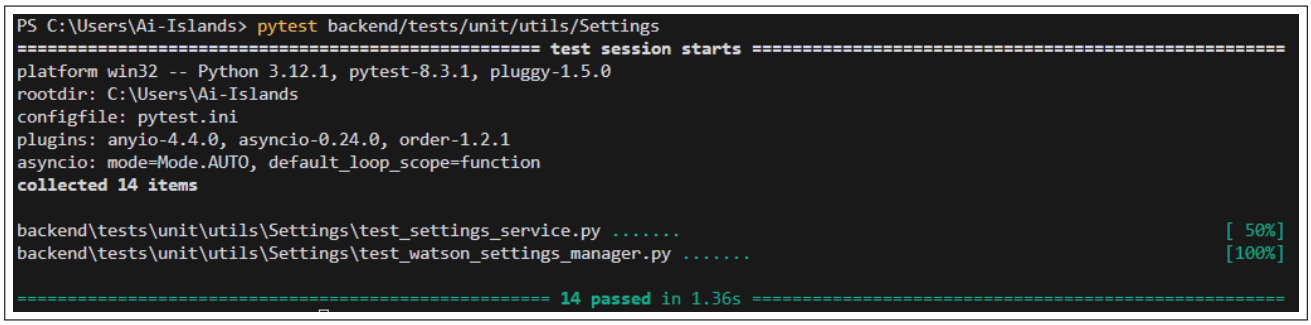


Figure G.5: Settings Functions Pytest

G.2 Integration API Tests

This section covers the test results for API integration tests in more depth, with results provided for the reader.

G.2.1 Testing - Settings Router

The settings router needs to be the first tested due to IBM Cloud settings required for the models used within the next sets of tests. Figure G.6 shows all test pass for setting and fetching all kinds of AI Islands settings, including tests for updating and retrieving Watson settings, RAG chunking configurations, hardware settings, and checking GPU availability.

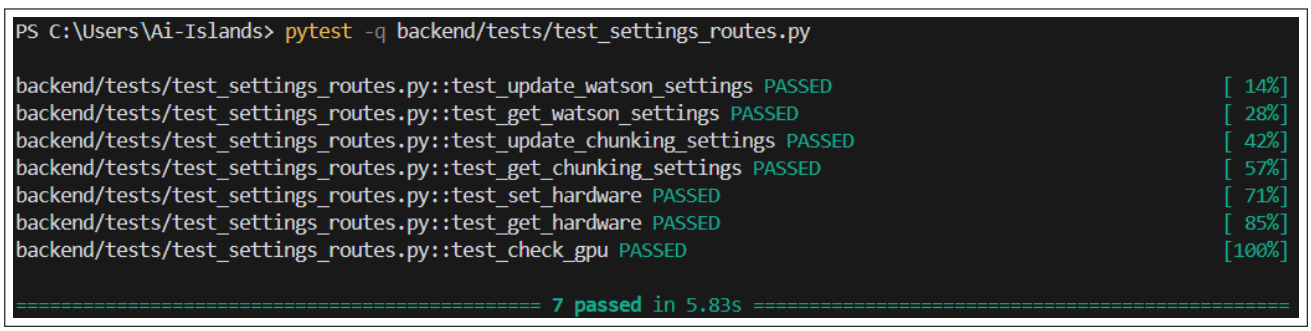


Figure G.6: Settings Router Pytest

G.2.2 Testing - Data Router

The data router tests needs to be run before the model router; since we test with a RAG context on the second inference in the model router test. Figure G.7 shows all of the necessary functionality for manipulating datasets within AI Islands. It includes tests for uploading, processing, checking status, retrieving information, verifying existence, and listing datasets, as well as getting available models.

```
PS C:\Users\Ai-Islands> pytest -q backend/tests/test_data_routes.py

backend/tests/test_data_routes.py::test_upload_dataset PASSED                    [ 14%]
backend/tests/test_data_routes.py::test_process_dataset PASSED                   [ 28%]
backend/tests/test_data_routes.py::test_dataset_processing_status PASSED         [ 42%]
backend/tests/test_data_routes.py::test_dataset_processing_info PASSED           [ 57%]
backend/tests/test_data_routes.py::test_datasets_processing_existence PASSED     [ 71%]
backend/tests/test_data_routes.py::test_list_datasets PASSED                     [ 85%]
backend/tests/test_data_routes.py::test_get_available_models PASSED              [100%]

============================================== 7 passed in 7.74s ===============================================
```

Figure G.7: Data Router Pytest

G.2.3 Testing - Model Router

The model router tests can be run as long as the settings and data router have been tested. It is necessary so that an API key is found for this test set. Figure G.8 shows all of the necessary functionality for interacting with models, including a double inference with a RAG configuration update; particularly useful to check if the RAG system is functioning as expected.

```
PS C:\Users\Ai-Islands> pytest -q backend/tests/test_model_routes.py

backend/tests/test_model_routes.py::test_get_model_info_index PASSED             [  8%]
backend/tests/test_model_routes.py::test_download_model PASSED                   [ 16%]
backend/tests/test_model_routes.py::test_get_model_info_library PASSED           [ 25%]
backend/tests/test_model_routes.py::test_load_model PASSED                       [ 33%]
backend/tests/test_model_routes.py::test_list_active_models PASSED               [ 41%]
backend/tests/test_model_routes.py::test_is_model_loaded PASSED                  [ 50%]
backend/tests/test_model_routes.py::test_inference PASSED                        [ 58%]
backend/tests/test_model_routes.py::test_configure_model PASSED                  [ 66%]
backend/tests/test_model_routes.py::test_inference_after_configuration PASSED    [ 75%]
backend/tests/test_model_routes.py::test_unload_model PASSED                     [ 83%]
backend/tests/test_model_routes.py::test_is_model_loaded_after_unload PASSED     [ 91%]
backend/tests/test_model_routes.py::test_delete_model PASSED                     [100%]

============================================ 12 passed in 38.54s ============================================
```

Figure G.8: Model Router Pytest

G.2.4 Testing - Library Router

The library router tests can be run at any time depending on whether the defined base model within the test file is present within the library JSON. It includes tests for fetching the model index, retrieving the library, getting model details, updating configurations, saving new models, and managing model IDs. Figure G.9 shows the manipulation of this file, including the mechanics behind configuration updates.

```
PS C:\Users\Ai-Islands> pytest -q backend/tests/test_library_routes.py

backend/tests/test_library_routes.py::test_get_full_model_index PASSED                [ 14%]
backend/tests/test_library_routes.py::test_get_full_library PASSED                    [ 28%]
backend/tests/test_library_routes.py::test_get_model_info_library PASSED              [ 42%]
backend/tests/test_library_routes.py::test_update_model_config PASSED                 [ 57%]
backend/tests/test_library_routes.py::test_save_new_model PASSED                      [ 71%]
backend/tests/test_library_routes.py::test_update_model_id PASSED                     [ 85%]
backend/tests/test_library_routes.py::test_restore_model_id PASSED                    [100%]

============================================================ 7 passed in 7.87s =============================================================
```

Figure G.9: Library Router Pytest

G.2.5 Testing - Playground Router

The playground router tests needs to be run after settings router in order for IBM services to operated. The test consists of creating a playground and changing the model management between inferencing. This is a good test to show how changing models can affect the output. It also tests stopping, deleting playgrounds, and unloading models.

```
PS C:\Users\Ai-Islands> pytest -q backend/tests/test_playground_routes.py

backend/tests/test_playground_routes.py::test_download_models PASSED                                                [  6%]
backend/tests/test_playground_routes.py::test_create_playground PASSED                                              [ 13%]
backend/tests/test_playground_routes.py::test_add_two_models_to_playground PASSED                                   [ 20%]
backend/tests/test_playground_routes.py::test_configure_chain_with_multiple_models PASSED                           [ 26%]
backend/tests/test_playground_routes.py::test_load_playground_chain_with_multiple_models PASSED                     [ 33%]
backend/tests/test_playground_routes.py::test_playground_inference_with_multiple_models PASSED                      [ 40%]
backend/tests/test_playground_routes.py::test_configure_chain_before_removing_model PASSED                          [ 46%]
backend/tests/test_playground_routes.py::test_remove_nlu_model PASSED                                               [ 53%]
backend/tests/test_playground_routes.py::test_get_playground_info PASSED                                            [ 60%]
backend/tests/test_playground_routes.py::test_configure_chain_with_single_model PASSED                              [ 66%]
backend/tests/test_playground_routes.py::test_reload_playground_chain_with_single_model PASSED                      [ 73%]
backend/tests/test_playground_routes.py::test_playground_inference_after_reload_with_single_model PASSED            [ 80%]
backend/tests/test_playground_routes.py::test_stop_playground_chain_with_single_model PASSED                        [ 86%]
backend/tests/test_playground_routes.py::test_delete_playground PASSED                                              [ 93%]
backend/tests/test_playground_routes.py::test_unload_and_delete_models PASSED                                       [100%]

============================================================ 15 passed in 77.64s (0:01:17) =============================================================
```

Figure G.10: Playground Router Pytest

G.3 User Acceptability Test

Table G.1 is the mark sheet used for the UAT. There were 4 different testers all of which executed each test case and marked their opinion out of 10, and indicated if they were successful.

ID	Description	Feedback /10	Passes /1
UAT01	Users successfully linked their IBM Watson accounts and performed inferences using IBM foundation models with configuration changes.	(37/40)	(4/4)
UAT02	Users successfully linked their IBM Watson accounts and performed inferences using IBM NLP models.	(32/40)	(4/4)
UAT03	Testers created custom workflows by chaining models for different use cases.	(27/40)	(4/4)
UAT04	Testers processed datasets and configured a watsonx LLM to use the dataset.	(36/40)	(4/4)
UAT05	Testers configured a watsonx LLM to the chatbot and commenced a conversation.	(25/40)	(4/4)

Table G.1: Test Scenarios

Appendix H

User & Deployment Manual

Welcome to the AI Islands user manual, a comprehensive guide to navigating and using the core features of the AI Islands application. This manual will walk you through the essential components of the platform, including the menu navigation, model management, playgrounds, and data processing ensuring that you can make the most out of your AI Islands experience.

In this guide, you will find detailed explanations and visual aids for working with AI models, customising configurations, managing datasets, and integrating models into your workflows. Whether you're managing AI models for inference, customising parameters, or processing datasets for enhanced performance, this manual covers all the basics to help you get started.

To run this application see the Deployment Manual in **Section H.7**

H.1 Navigation

The app shell navigation consists of two menus, a main menu fly-out and a helper menu drop-down.

H.1.1 Menu Bar

The main menu fly-out consists of 5 main areas within AI Islands; AI Index, Library, Playground, Data Management, and Settings.

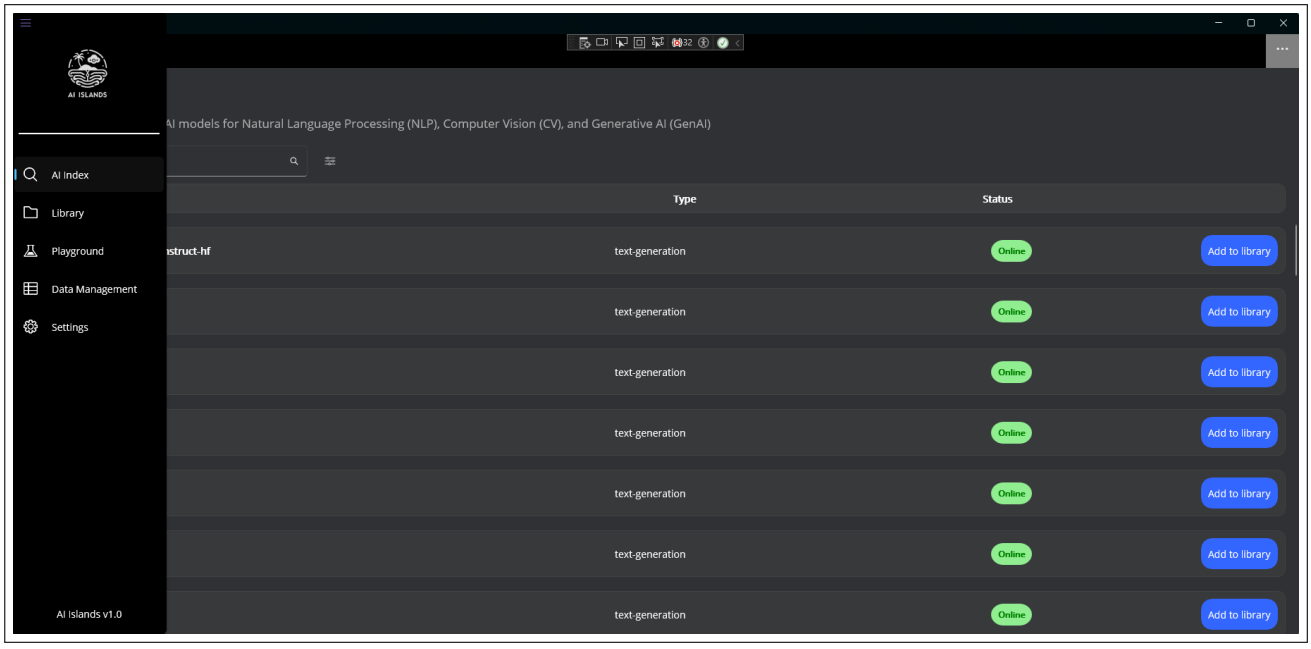


Figure H.1: AI Islands Main Menu

H.1.2 Helper Menu

This drop-down menu consists of tutorial and model website links. This is particularly useful for users to access their IBM Cloud account if AI Islands has prompted them with account issues.

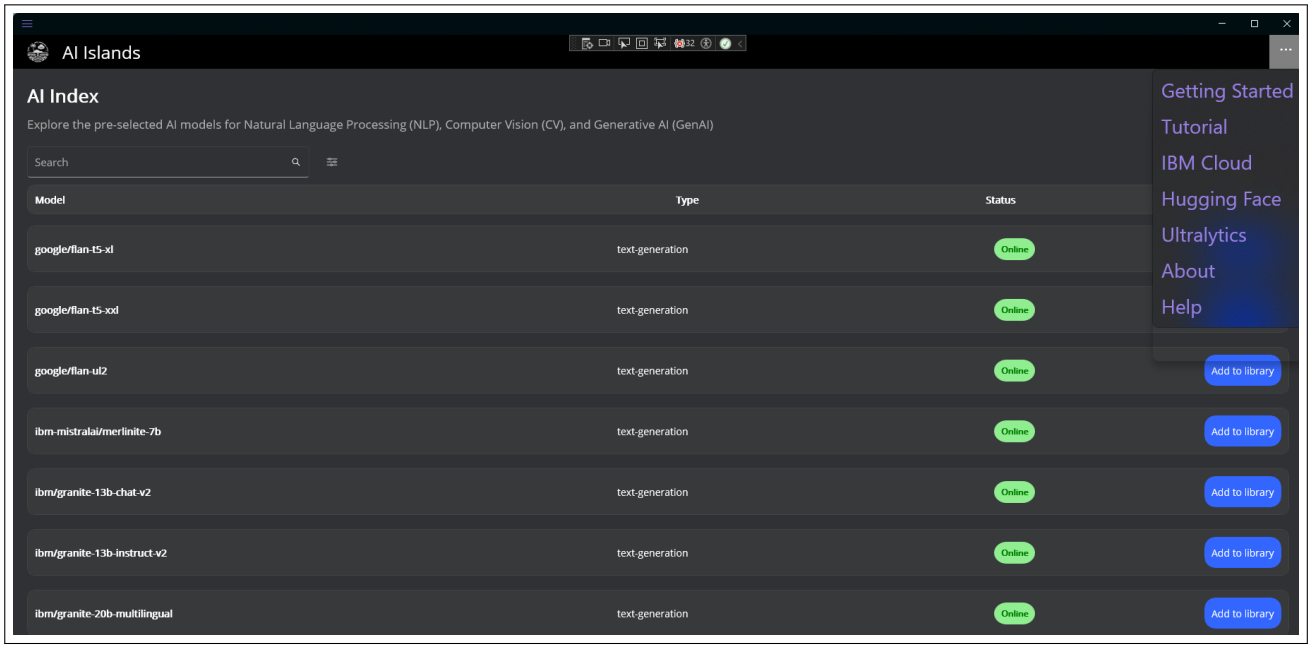


Figure H.2: AI Islands Helper Menu

H.2 AI Index

H.2.1 AI Index

The model index is where users can browse and read information about certain models. They are provided with a filtering system, see Figure H.4, for quick model access. Users download models they wish to use from this index.

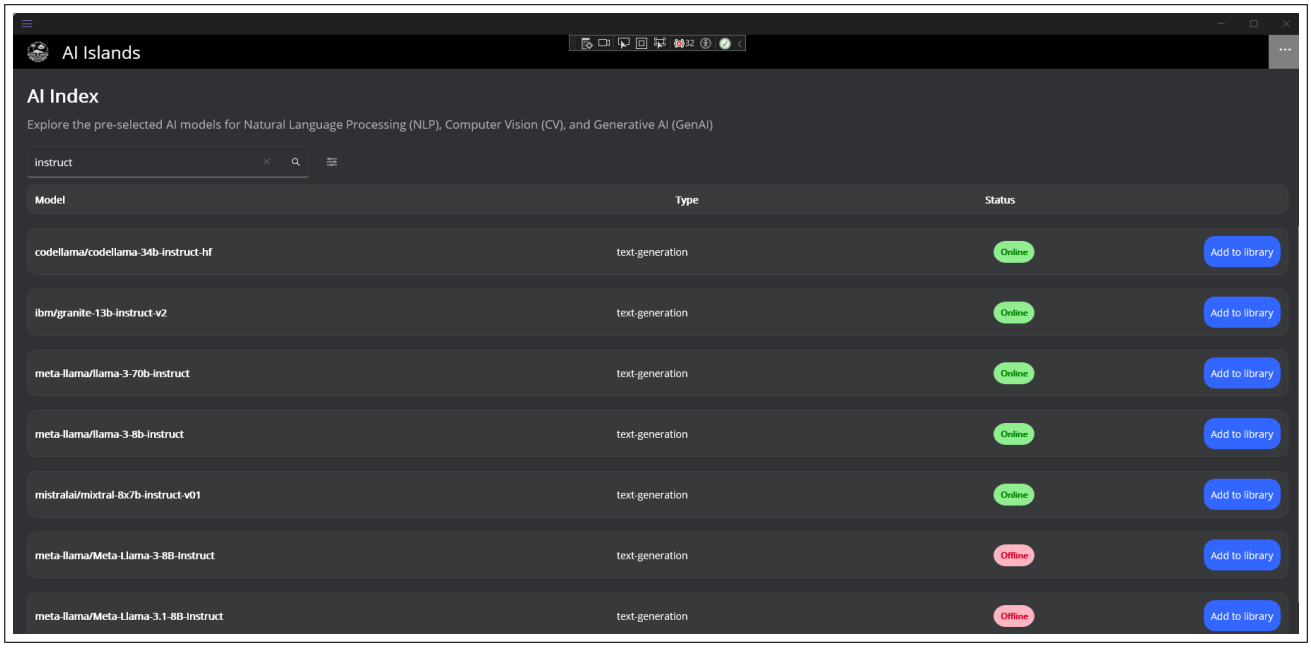


Figure H.3: AI Islands AI Index

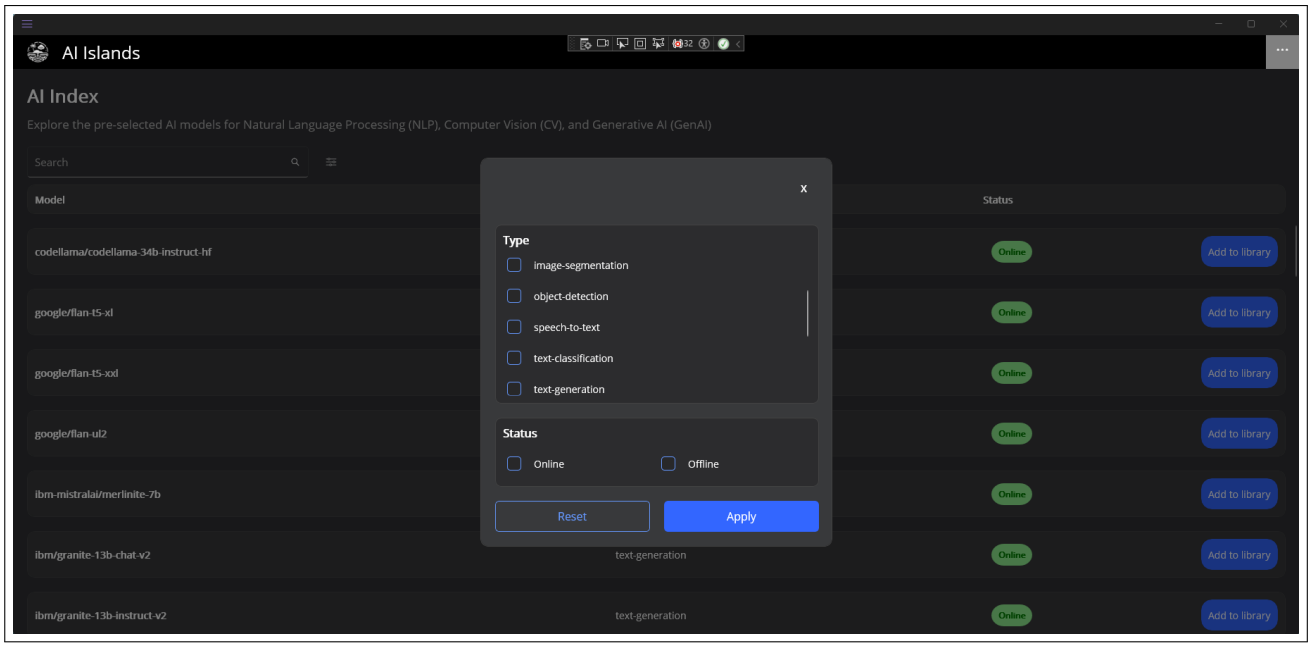


Figure H.4: AI Islands Model Index Filtering

H.3 Library

H.3.1 Library Index & Information

The Library in Figure H.5 shows all downloaded models. Users can load and unload models from here. The can also delete models from their local files using the bin icon. Users can observe important model information such as the base model and whether or not a configuration has occurred, different to the base model.

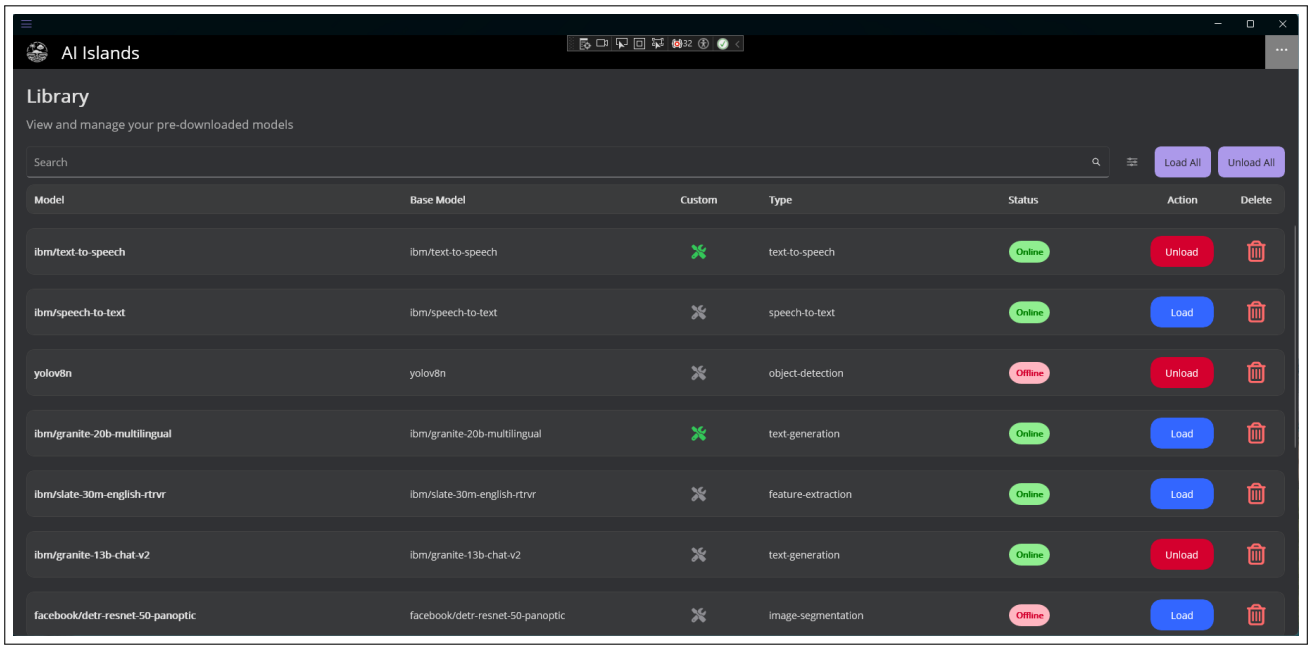


Figure H.5: AI Islands User Library Index

Users can click on a model, bringing them to a tabbed navigation system, consisting of Info, Inference, Fine Tune, Configuration, and API Access. They can view model information on this page, see Figure H.6

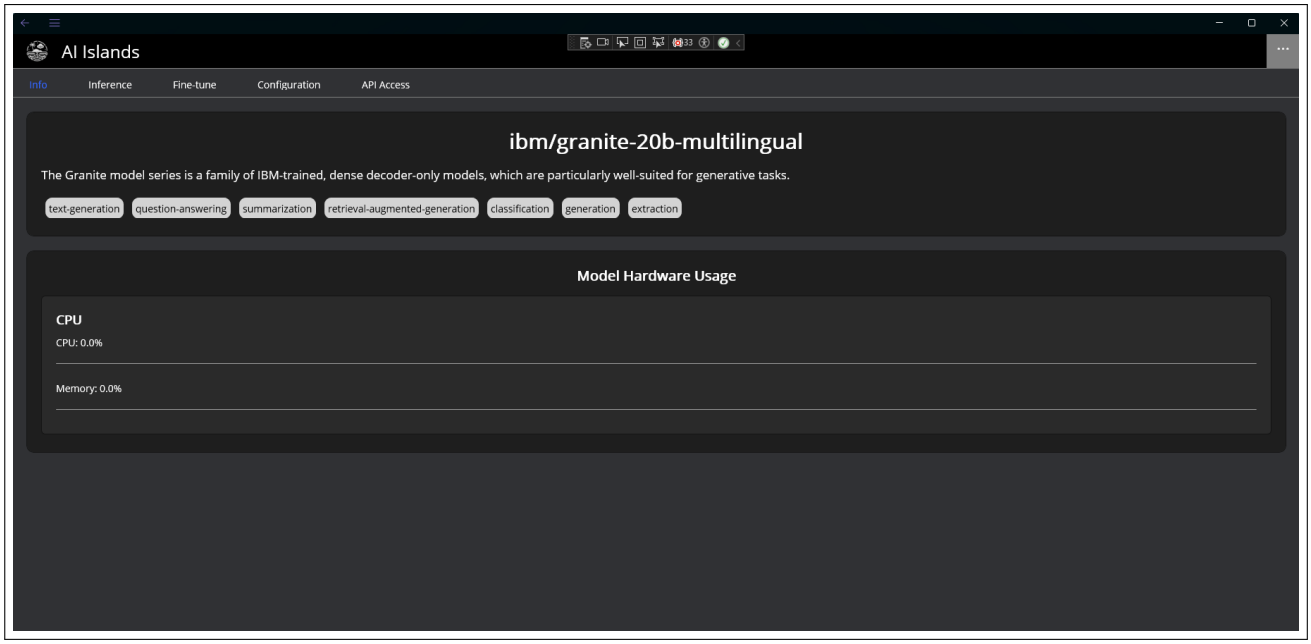


Figure H.6: AI Islands Model Information View

H.3.2 Inference

Users can inference models on the page shown in Figure H.7, provided the user had loaded the model in their library page.

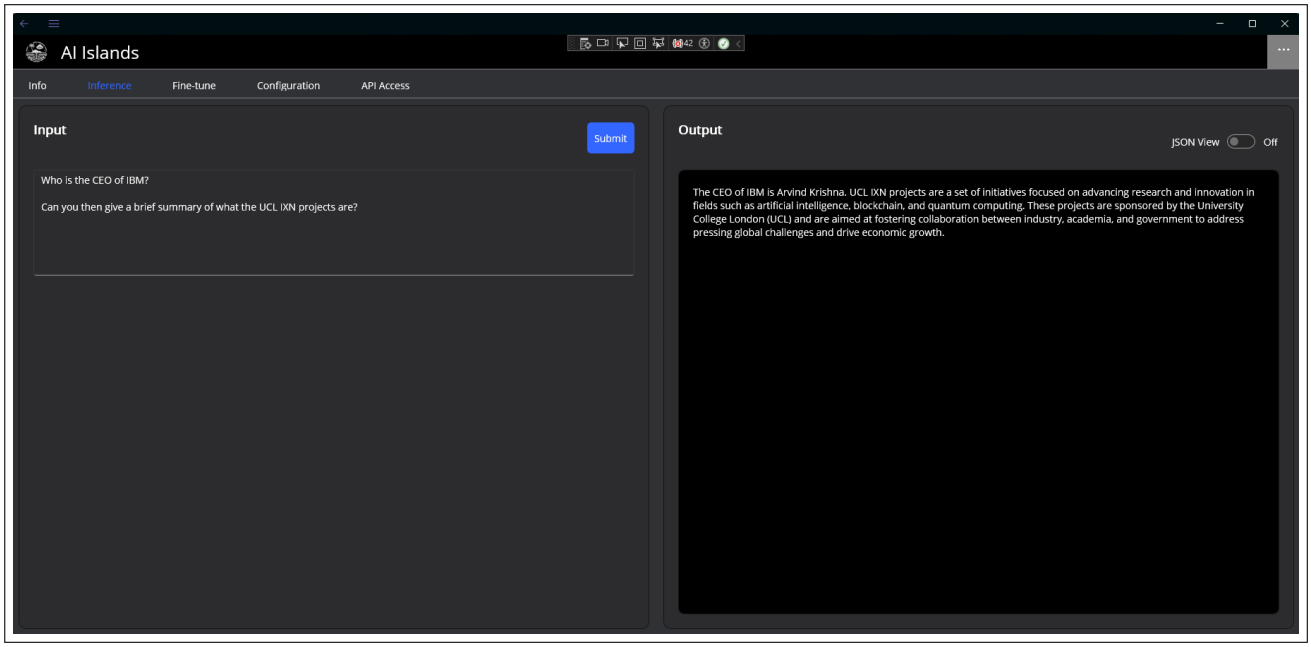


Figure H.7: AI Islands Library Model Inference

H.3.3 Configuration

Users can customise their models by adjusting their configurations, as seen in the sample page in Figure H.8. Additionally, Figure H.9 shows the necessary buttons which restore original configurations, save configurations to current, or save configurations under a new model name.

Figure H.8: AI Islands Configuration Sample Page

H.3.4 Chat Bot

Users can configure their LLMs to a chat bot instead of a standard inference, as shown in Figure H.9.

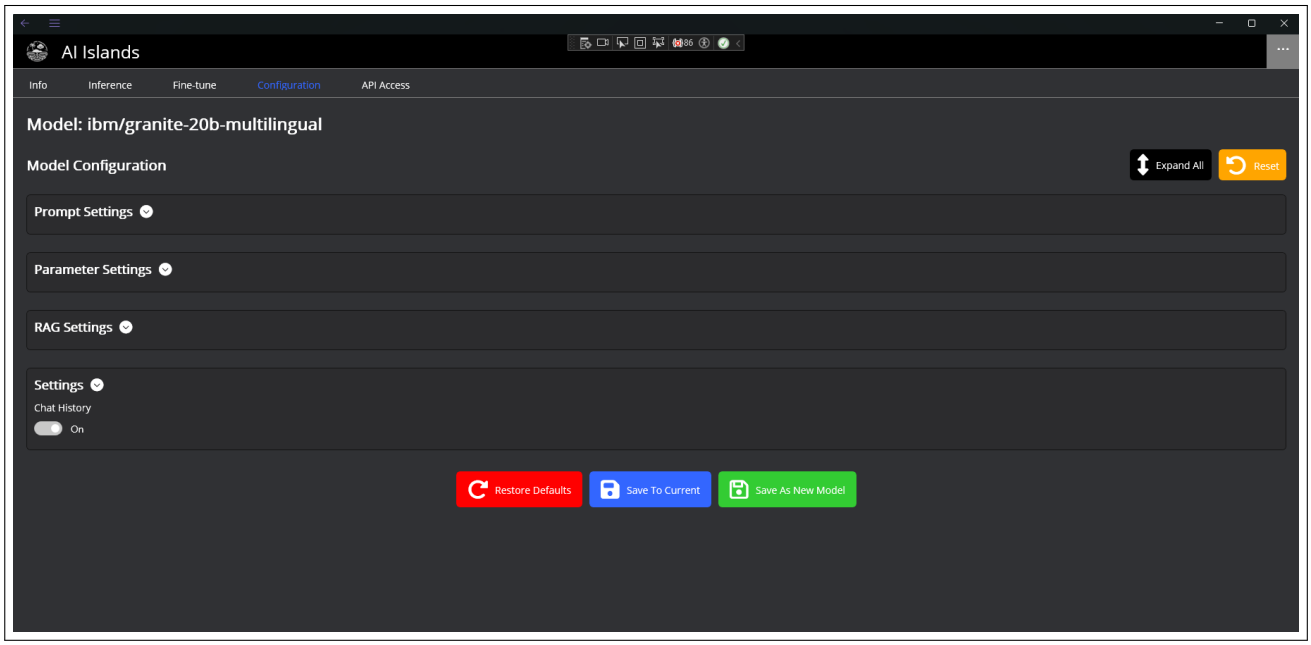


Figure H.9: AI Islands Configuration for Chat Bot Option

Users can return to the inference page seen in Figure H.10, and use the chat bot feature provided they set their chat history to *on* in the models' configuration. In this case, the user updated the models' RAG settings to include a dataset context base. This same dataset is seen in the processing guide in **Section H.5**.

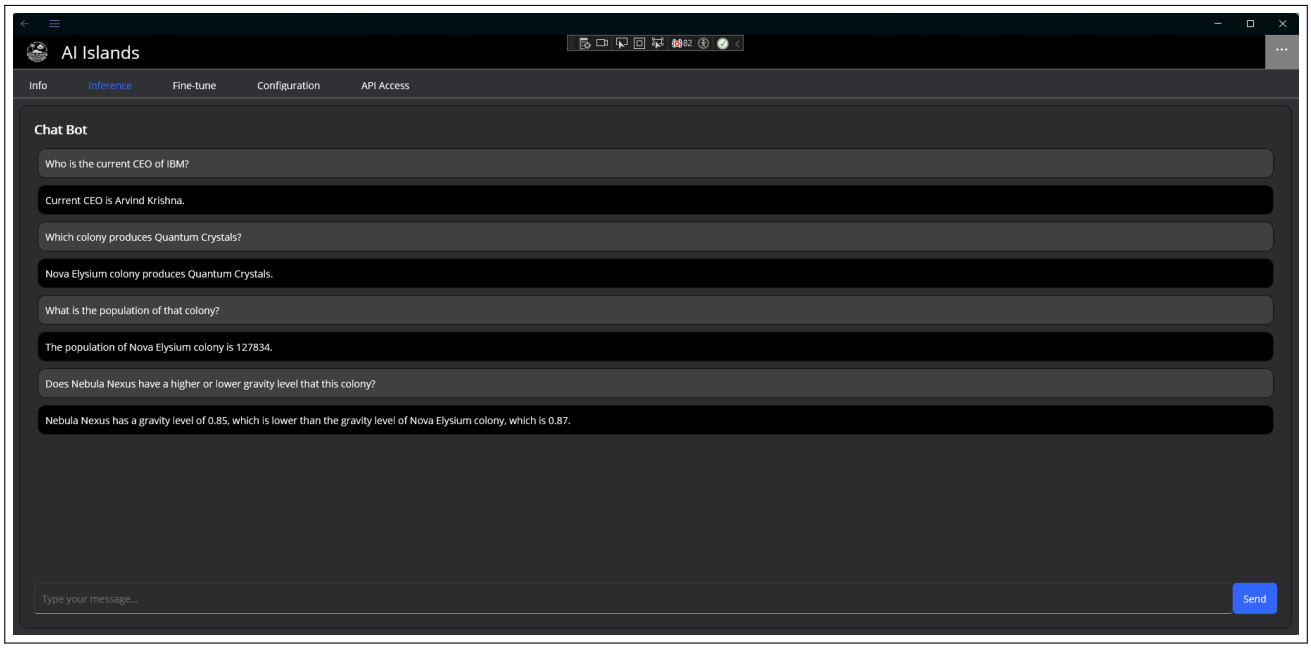


Figure H.10: AI Islands Chat Bot Feature

H.3.5 API Access

API Access is available for implementing models within an external application. The endpoints are listed in this page, as show in Figure H.11.

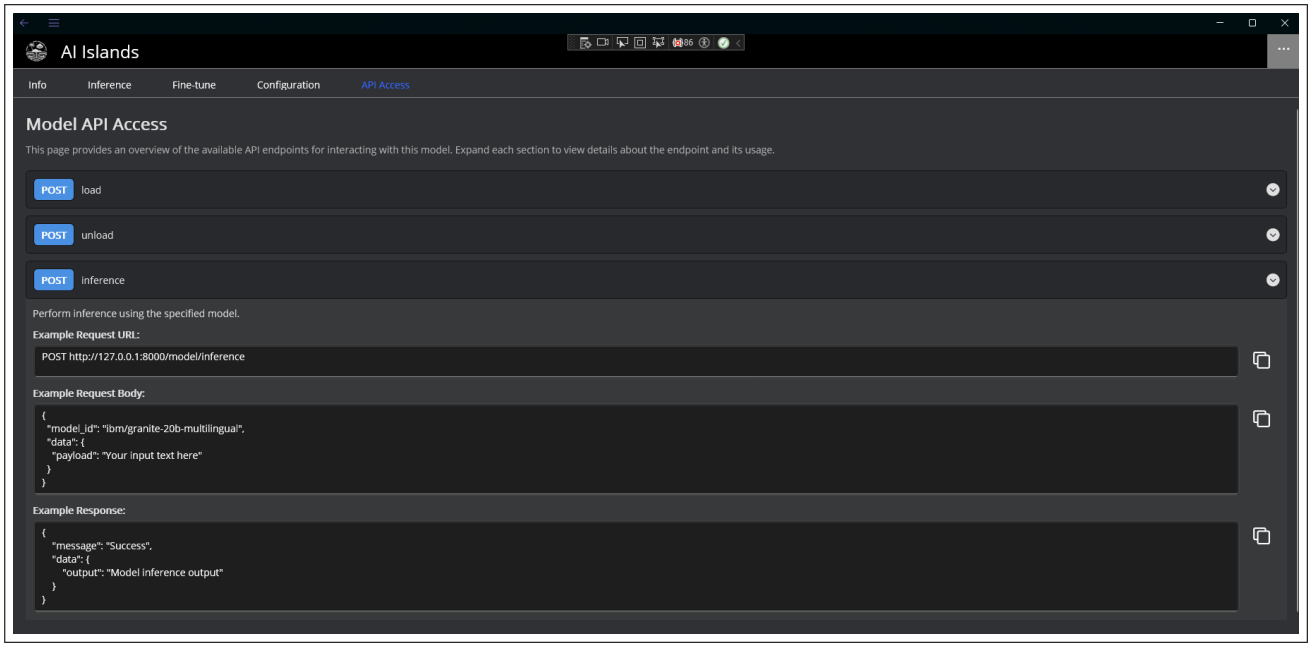


Figure H.11: AI Islands Model API Access

H.4 Playground

H.4.1 User Playgrounds

Users can view all of their playgrounds, as shown in Figure H.12. They can create and edit playgrounds, as well as delete them.

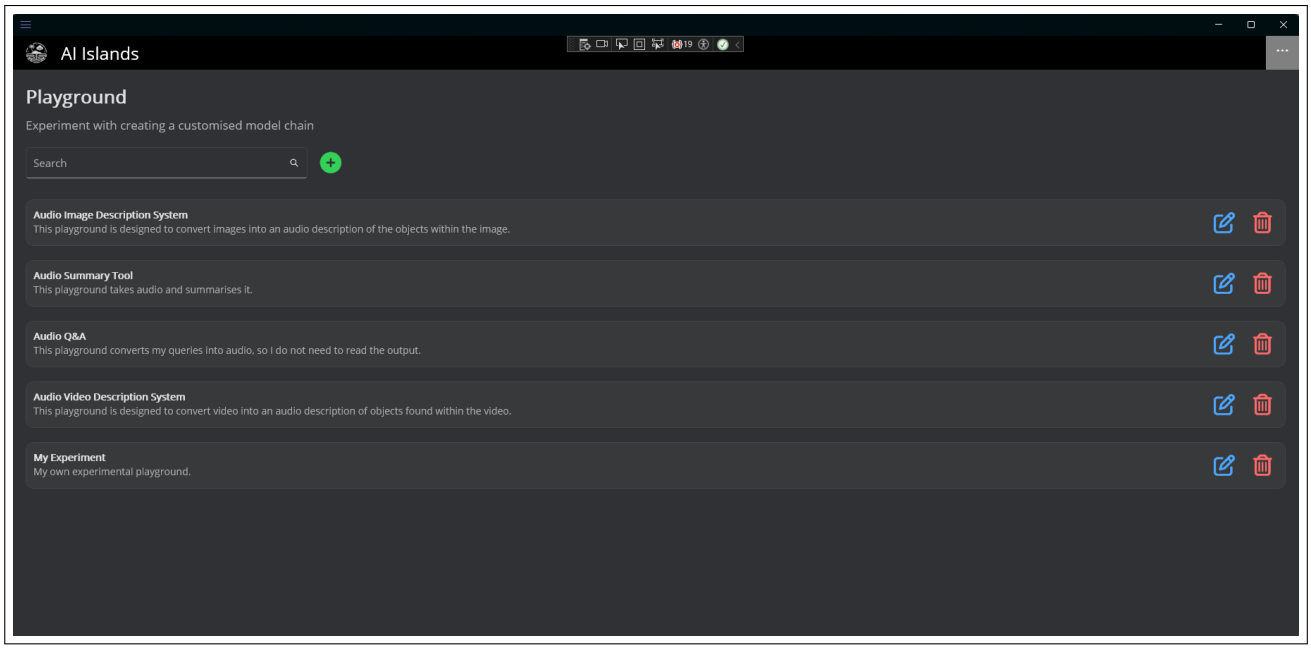


Figure H.12: AI Islands User Playgrounds

H.4.2 Playground Model Management

Users manage models in their playground by first adding the models themselves in Figure H.13, and later configuring their order in Figure H.14.

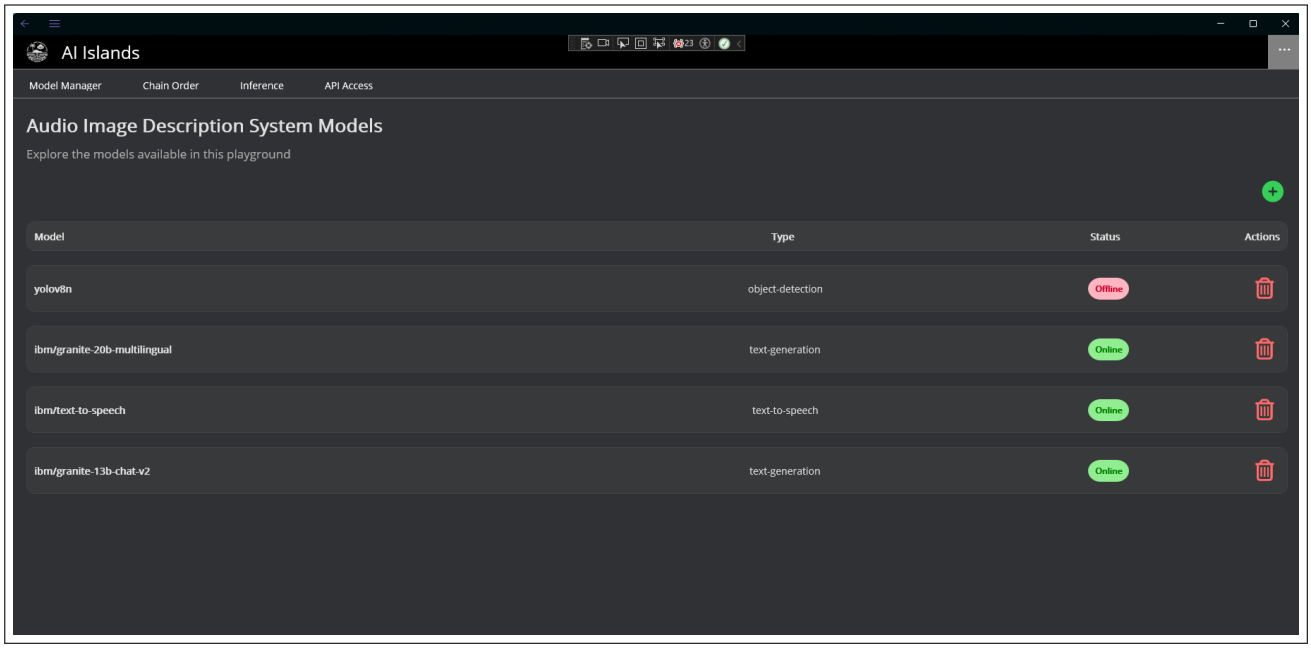


Figure H.13: AI Islands Playground Model Manager

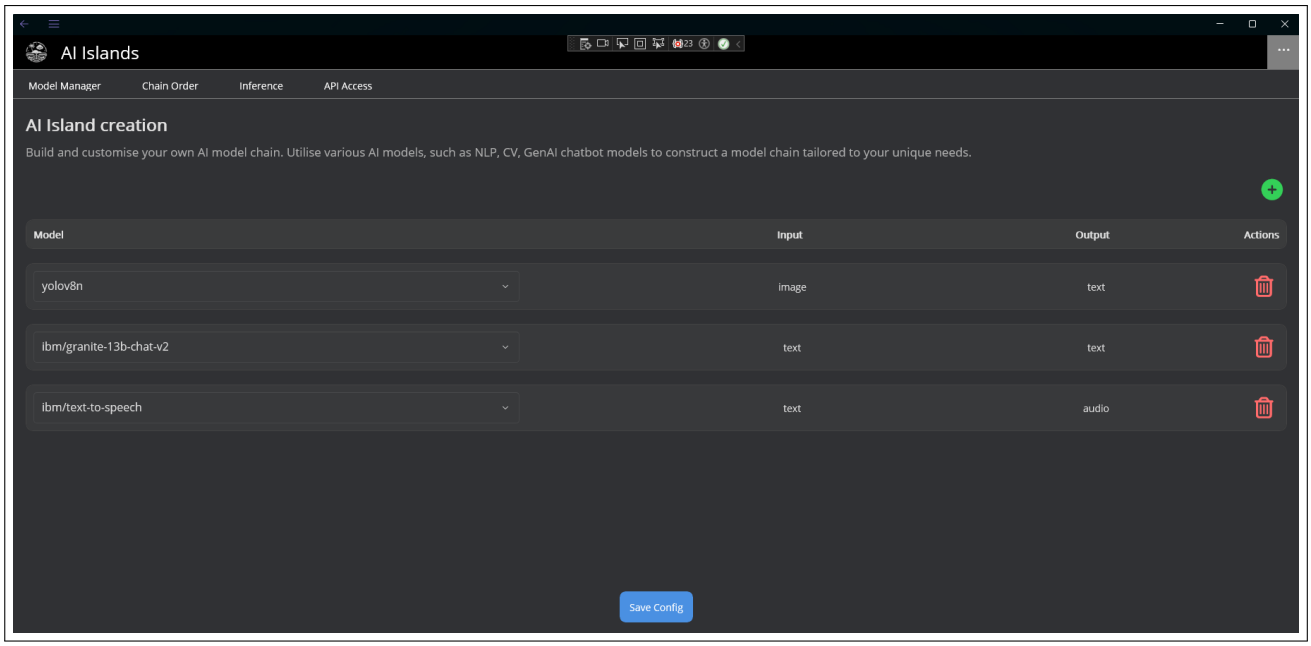


Figure H.14: AI Islands Playground Model Chain Ordering Configuration

H.4.3 Playground Inference

Users can inference the model chain they define within their playground, as shown in Figure H.15 with an image file input and an audio file output.

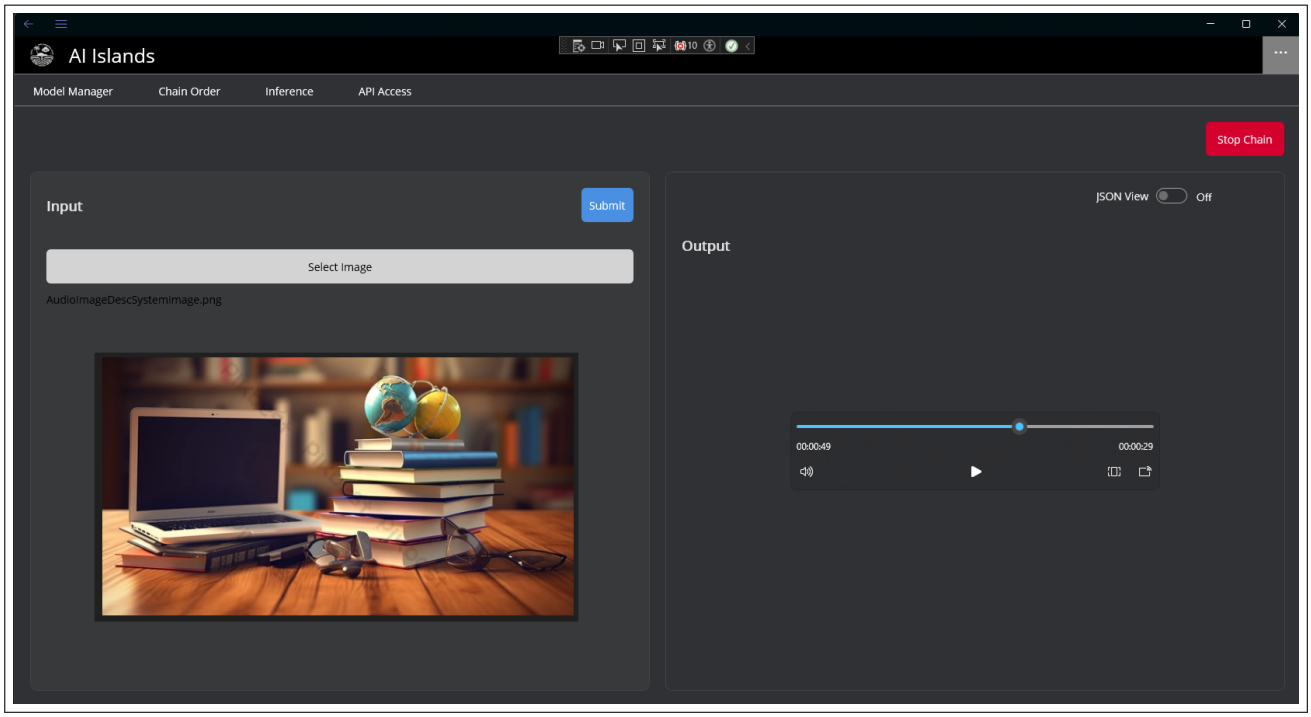


Figure H.15: AI Islands Playground

H.5 Data Management

H.5.1 Dataset Upload

Users can upload their own datasets provided they are in valid formats (.csv, .txt, .md, .docx, .doc, .pdf), as shown in Figure H.16.

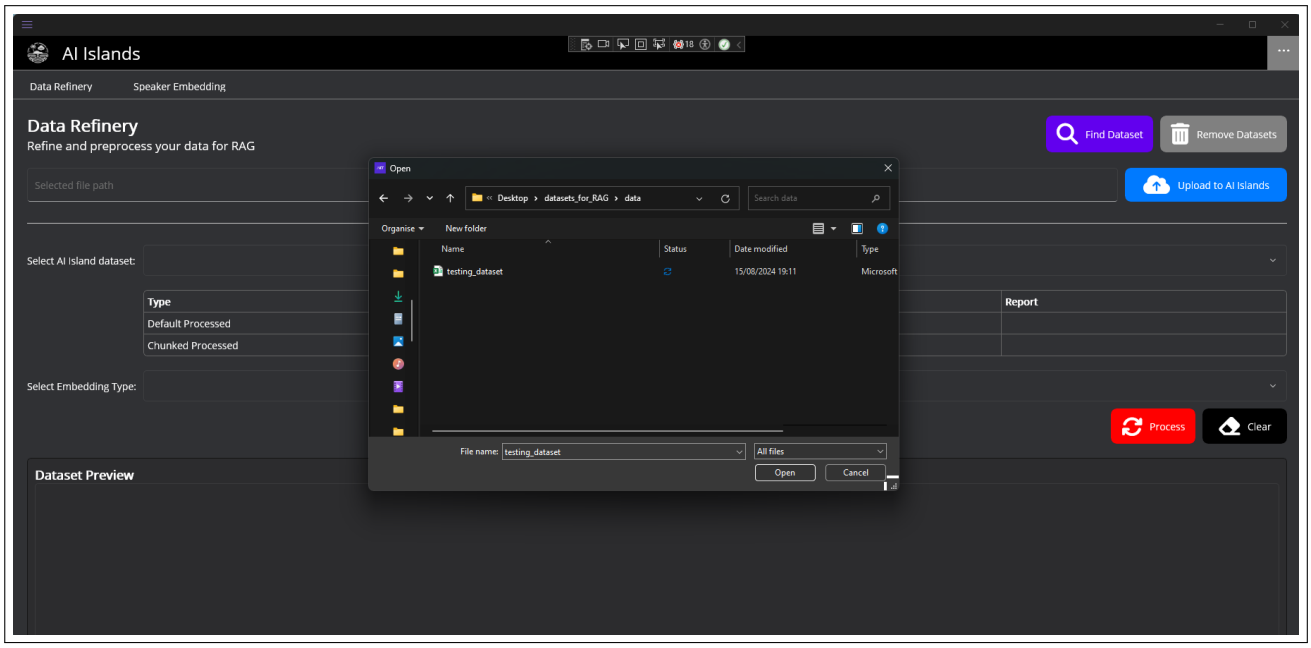


Figure H.16: AI Islands

H.5.2 Dataset Processing

Users can select uploaded datasets and preview them, as shown in Figure H.17. Note the process button is red; showing a valid processing configuration is yet to be selected by the user.

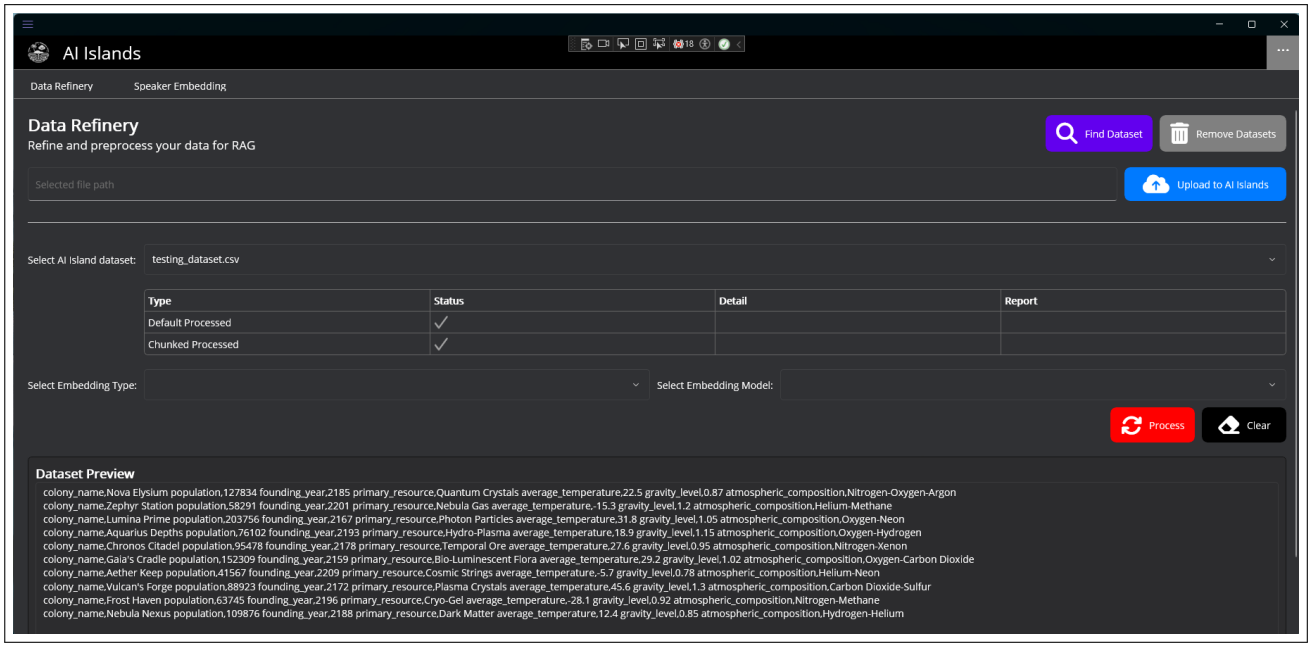


Figure H.17: AI Islands

Users can select their embedding type and specific embedding model as shown in Figure H.18. Note the process button is now green; suggesting the dataset is now ready fro processing.

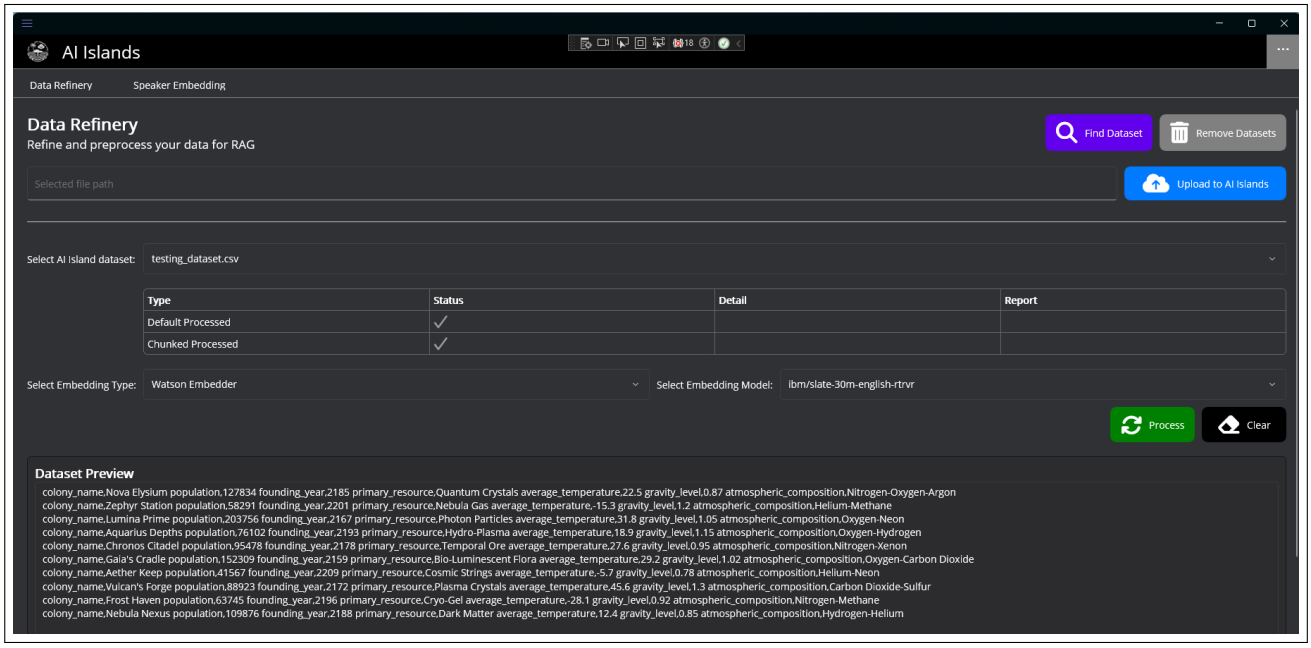


Figure H.18: AI Islands

After processing is finished, users can view reports by clicking the links, shown in Figure H.19.

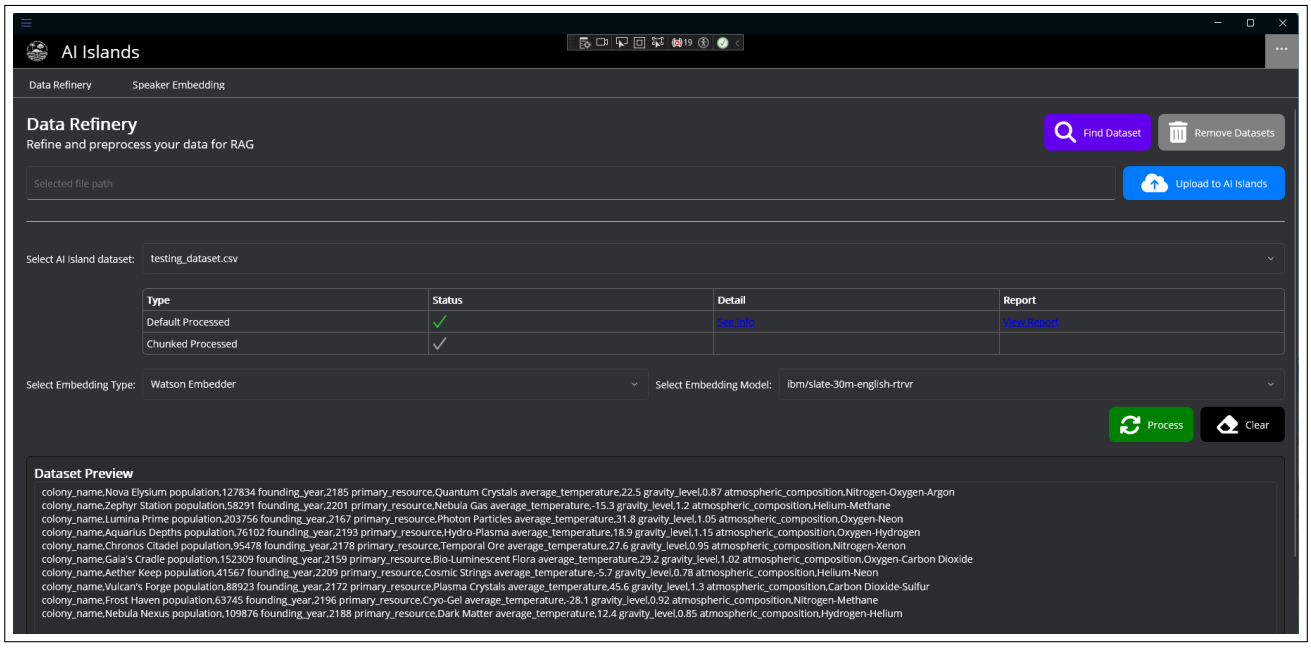


Figure H.19: AI Islands

H.5.3 Dataset Processing Warning

A user has the option to use watsonx embedders. The user will need to ensure that they have set a valid API key and project ID, otherwise they will be notified of the issue, as shown in Figure H.20.

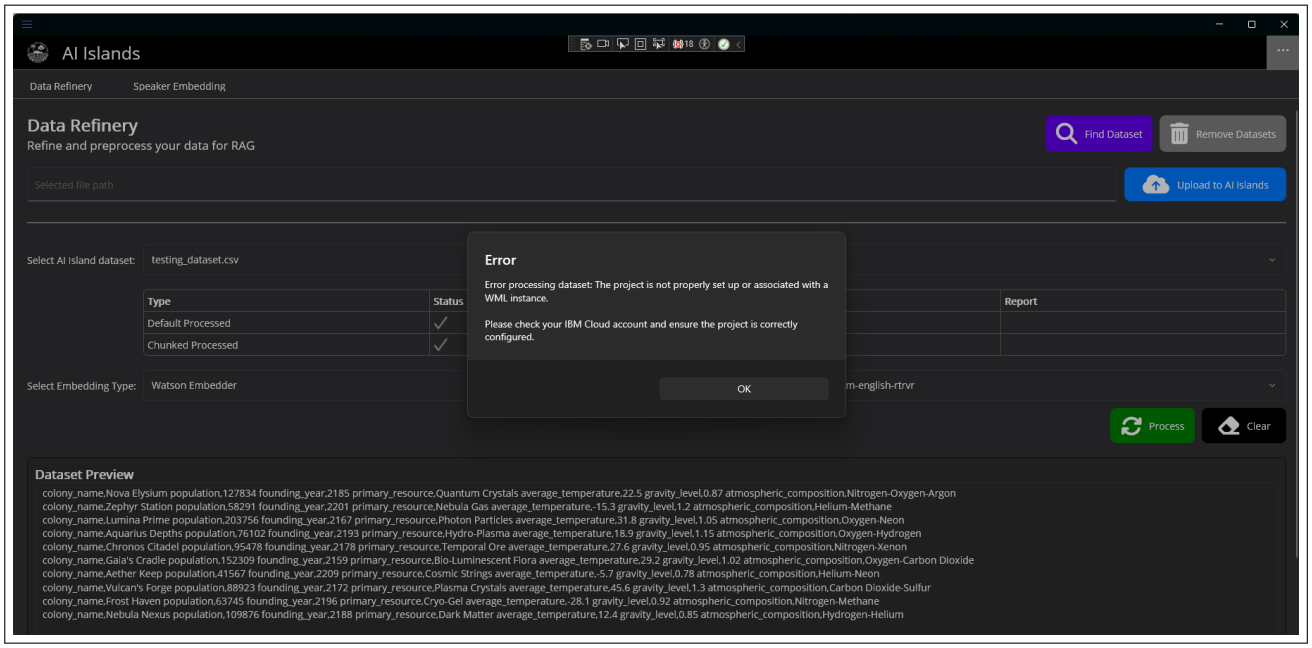


Figure H.20: AI Islands

H.6 Settings

H.6.1 IBM Cloud Account Settings

Users can edit their API key, location, and project ID in the watson cloud account settings, as shown in Figure H.21.

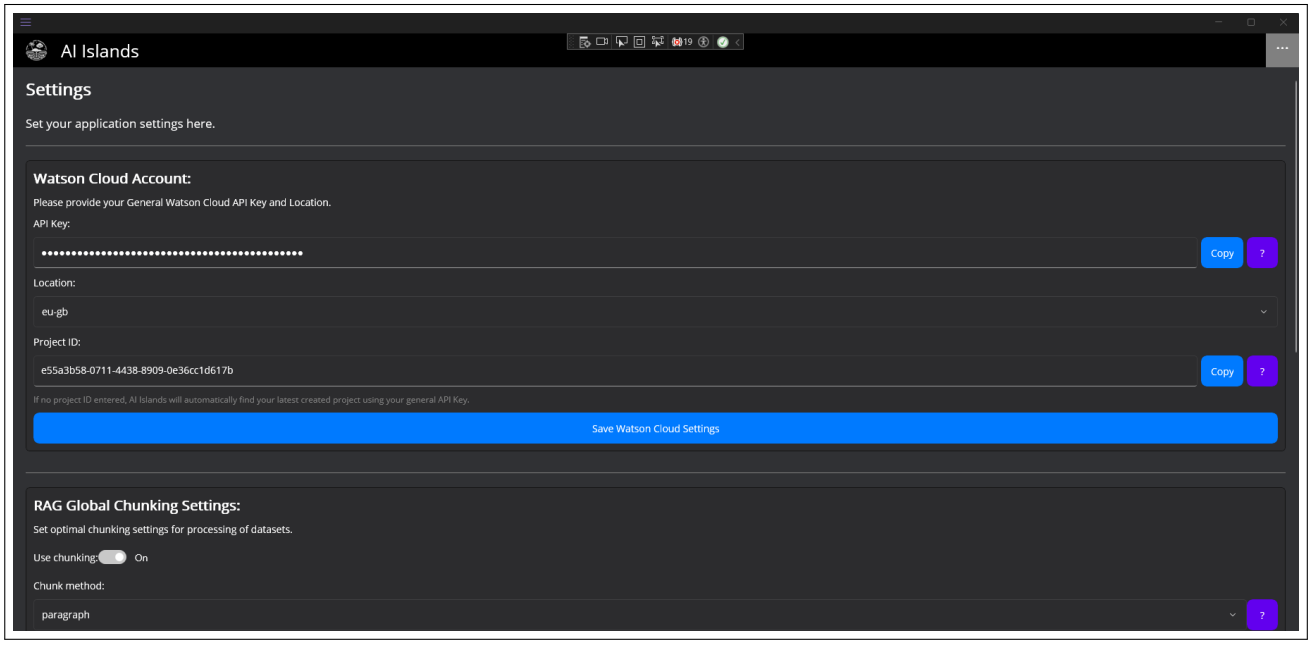


Figure H.21: AI Islands IBM Cloud Account Settings

H.6.2 RAG Settings

Users can edit RAG settings; either turn chunking on or off. If users turn this feature on, they can edit various child settings. Users can click '?' buttons to show information about a particular setting, as shown in Figure H.22

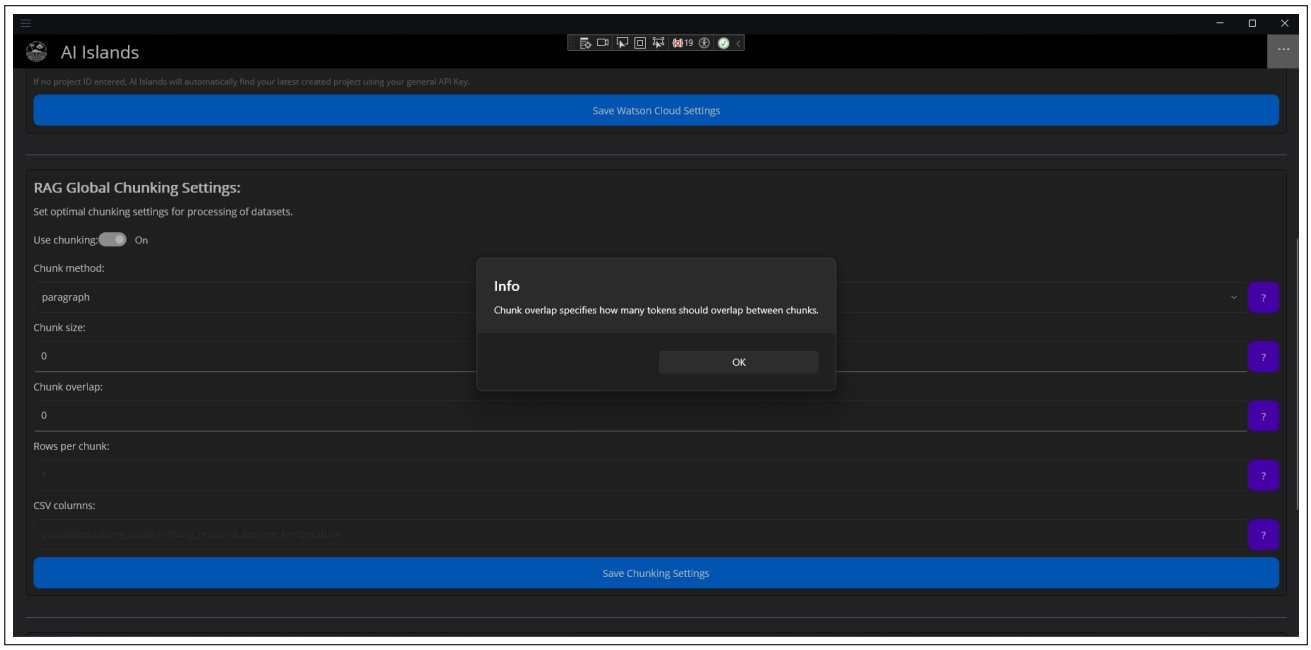


Figure H.22: AI Islands RAG Settings

H.7 Deployment

These are the steps required to access and set up AI Islands. As the current version has not been compiled, the necessary setup instructions are as followed.

Prerequisites:

Visual Studio Community 2022; Visual Studio installed with the .NET MAUI workload selected: `https://dotnet.microsoft.com/en-us/learn/maui/first-app-tutorial/install`. **Python**; The latest stable version of Python must be installed: `https://www.python.org/downloads/` **GitHub Repository**; To run the application, clone or download the AI Islands project from GitHub: `https://github.com/JJBZZRD/Ai-Islands`.
(*__To run IBM specific services in the application refer to Chapter 6.__*)

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.

If repository access is unavailable, a copy of the README file can be found in the AI Islands source code in supplementary material. A PDF print out is also provided.