Productionizing SPIRA's model trainer system

Lucas Quaresma Medina Lam
Roberto Oliveira Bolgheroni

Capstone project
Presented to the Discipline
MAC0499

Supervisors:
Prof. Dr. Alfredo Goldman
M.Sc. Renato Cordeiro Ferreira

São Paulo, December 2024

Contents

Abstract

BOLGHERONI, R. O., LAM, L. Q. M. Productionizing SPIRA's model trainer system. Thesis (BSc.) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

SPIRA is an intelligent system designed to provide pre-diagnoses of respiratory insufficiency through speech analysis. Although the model trainer system underwent a redesign in 2023, it was not production-ready. This project aimed to implement the redesign by introducing a comprehensive automated test suite, enabling Continuous Integration and Continuous Deployment (CI/CD) practices via automated pipelines, and preparing the system for integration with a scheduling platform. The system was rewritten using Test-Driven Development (TDD) practices, with the hexagonal architecture pattern applied to enhance code design and modularity. Additionally, the training pipeline's steps were decoupled into independent executable applications, and CI/CD pipelines were implemented on GitHub. These advancements collectively lay the groundwork for future system development and seamless integration with scheduling systems.

Keywords: SPIRA, Intelligent Systems, CD4ML, Automated Tests, Hexagonal Architecure

Resumo

BOLGHERONI, R. O., LAM, L. Q. M. Produtizando o treinador de modelos do SPIRA. Thesis (BSc.) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

SPIRA é um sistema inteligente projetado para fornecer pré-diagnósticos de insuficiência respiratória por meio da análise de voz. Embora o Sistema Treinador de Modelos tenha passado por um redesign em 2023, ele ainda não estava pronto para produção. Este projeto teve como objetivo implementar o redesign introduzindo uma cobertura abrangente de testes automatizados, habilitando práticas de Integração Contínua e Entrega Contínua (CI/CD) por meio de pipelines automatizadas e preparando o sistema para integração com uma plataforma de agendamento. O sistema foi reescrito utilizando práticas de Desenvolvimento Orientado por Testes (TDD), com o padrão de arquitetura hexagonal aplicado para melhorar o design do código e sua modularidade. Além disso, as etapas da pipeline de treinamento foram desacopladas em aplicações executáveis independentes, e pipelines de CI/CD foram implementadas no GitHub. Esses avanços estabelecem as bases para o desenvolvimento futuro do sistema e a integração com sistemas de agendamento.

Palavras-chave: SPIRA, Sistemas Inteligentes, CD4ML, Testes Automatizados, Arquitetura Hexagonal

Chapter 1

Introduction

The SPIRA project emerged as an initiative to develop a software system capable of performing pre-diagnosis of respiratory insufficiency through speech analysis based on Machine Learning (ML) models. Initially built in 2021 (Casanova et al., 2021), SPIRA's training system underwent a redesign led by Daniel Lawand, aiming to apply good practices in software engineering, with the ultimate goal of implementing a continuous delivery pipeline. Although the redesign was a success, the code base wasn't in a production-ready state. Particularly, it had no automated tests, and the pipeline wasn't finalized (Lawand, 2023).

The goal of this project was to productionize SPIRA's model trainer system. For that, the aim was to provide comprehensive automated tests and to finish the continuous delivery pipeline. Productionizing the model trainer system makes the process of training and deploying new models reproducible, safer, and faster through automation. It promotes shorter development cycles, enhances maintainability and the software quality of the project, and enables continuous improvement of the model (Sato et al., 2019).

The following chapters present the specifics of the project. Chapter 2 discusses the background required for understanding SPIRA, it's previous contributions, and the upcoming topics in this document. Chapter 3 reviews the key concept of Automated Testing, and elaborates about its relevance to the project's goal. Chapter 4 and Chapter 5 describe the SPIRA project and the latest version of the SPIRA Model Trainer System, named V2, which stood as a basis for this project. Chapter 6 describes this project's proposed version for the Model Trainer System, named V3. Chapter 7 discusses the main contributions presented to the SPIRA model trainer system by this project. Finally, Chapter 8 reviews the key topics discussed in this document and lays out the next steps in the SPIRA project's road-map.

Chapter 2

Concepts

This chapter introduces the foundational knowledge necessary for understanding the proposal produced in this capstone project. It will encompass fundamental concepts in engineering and machine learning (ML) that are crucial for comprehending the intelligent systems and processes discussed further in this document.

The topics to be covered include Intelligent Systems, Model Training, Model Serving, Machine Learning Engineering, Continuous Delivery for Machine Learning (CD4ML), and the Hexagonal Architecture Pattern. These concepts lay the groundwork for the technical aspects and methodologies involved in developing and deploying ML-based systems.

2.1 Intelligent Systems

Intelligent Systems are systems in which the intelligence evolves and improves over time, particularly when its intelligence improves by incorporating feedback of how users interact with the system (Hulten, 2018).

Intelligent Systems connect users to artificial intelligence to achieve meaningful objectives. To connect the user to the intelligence of the system – given it uses machine learning models – two different processes must take place: the training of the machine learning models, and their subsequent serving. These two concepts are further described in the following sections.

2.1.1 Model Training

Training is an essential process that involves using algorithms to learn patterns in available data (Amazon, 2024). This process results in the creation of an ML model, which can make predictions or decisions based on input data. Model training can be iterative. It involves adjustments to algorithm parameters to optimize the model's performance regarding the available data.

2.1.2 Model Serving

Serving refers to the process of making trained models available for use in real-world applications (MarkovML, 2023). Once a model is trained, it needs to be deployed so that it can receive input data and generate predictions. Model serving involves setting up the underlying infrastructure to host the model and handle incoming requests. It also includes monitoring the model's performance in production, as well as ensuring scalability and reliability.

2.2 Machine Learning Engineering

Machine Learning Engineering involves a set of practices and techniques for the development, implementation, maintenance, and updating of ML-based systems (Wilson, 2022). This includes strategic project planning, selection of suitable algorithms, data collection, data preparation, model deployment, and continuous monitoring. This approach allows stakeholders to benefit from adopting machine learning techniques.

2.3 Continuous Delivery for Machine Learning

Continuous Delivery for Machine Learning (CD4ML) is a software engineering approach in which a cross-functional team produces machine learning applications in small and safe increments that can be reproduced and reliably released at any time, in short adaptation cycles (Sato et al., 2019).

CD4ML is the application of the Continuous Delivery concept for Machine Learning applications. It differs from the original concept because ML applications have three axes of changes: data, model, and code. To achieve Continuous Delivery in these systems, changes in any of the three axes have to be addressed.

The concept is materialized in the form of a pipeline that streamlines processes such as data collection and preparation, model training and validation, performance evaluation, and automated deployment. This enables rapid updates and improvements to models, thus enhancing the effectiveness and relevance of ML systems.

2.4 Hexagonal Architecture Pattern

Hexagonal architecture pattern, also known as the Ports and Adapters pattern, is a software design paradigm that emphasizes the separation of business logic from external dependencies. Its primary goal is to create maintainable and flexible applications by isolating the core logic from external dependencies such as databases, APIs, frameworks, and user interfaces. This approach promotes adaptability, as changes to external systems do not imply significant impacts on the application's core functionality.

The pattern envisions an application as a collection of concentric layers, with the innermost layer, the domain layer, encapsulating the domain business rules, algorithms and logic. This core is surrounded by ports that define explicit entry points for communication. External systems interact with the core only through these entry points, and the core package delegates operations to adapters through the abstract ports, employing dependency inversion. By adhering to these principles, the architecture promotes a clean and testable design.

The structure of hexagonal architecture can be visualized as a hexagon, where each side represents a distinct adapter interfacing with the core through predefined ports. Adapters can include user interfaces, frameworks, databases, or external APIs, each implementing specific ports relevant to core goals. This separation allows independent implementation, testing, and evolution of the core and its adapters. For instance, a database adapter's framework can be swapped, or the database provider can be replaced without impacting the domain layer, highlighting the flexibility and resilience of this approach. Figure 2.1 illustrates an example of hexagonal architecture.

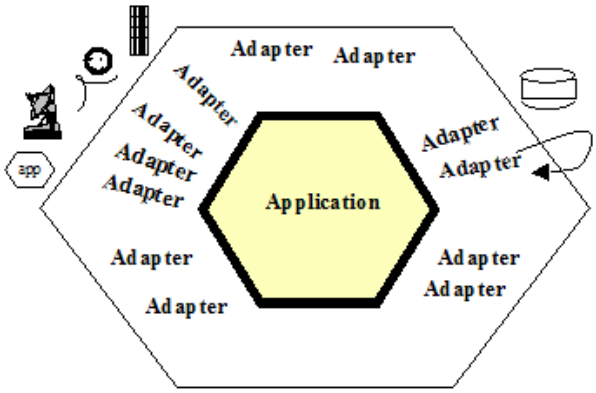

Figure 2.1: *Hexagonal Architecture diagram illustrating the use of adapters to connect the core application with external systems. Adapted from Cockburn (2008).*

Another key advantage of hexagonal architecture is its ability to simplify testing. By decoupling the domain logic from external systems, developers can test the core functionality in isolation, free from the complexities of integrating with external dependencies, which promotes agility (Cockburn, 2008).

Chapter 3

Automated Testing

Automated testing plays a crucial role in improving the quality and reliability of machine learning and intelligent systems. Despite the inherent challenges posed by non-deterministic behaviors, such systems can benefit from automated test suites, as outlined in (Sato et al., 2019):

1. **Validating data**: Automated tests can ensure that the training data adheres to the expected schema and properties. This includes verifying data integrity, detecting anomalies, and validating the accuracy of engineered features.

2. **Validating component integration**: These tests assess the interactions between different components of an intelligent system. They ensure that data flows correctly across modules and that integration points function as intended, reducing the risk of system-wide failures.

3. **Validating model quality, bias, and fairness**: By evaluating metrics such as precision, recall, and accuracy, automated tests can confirm that new models meet or exceed a defined performance baseline. Additionally, fairness tests can analyze model performance across specific data slices, identifying potential biases and ensuring equitable outcomes for all user groups.

4. **Components integration and software validation**: Intelligent systems contain not only machine-learning focused code, but traditional software and components that also benefit from automated testing practices. In particular, validating the integration of these components and the software correctness proves valuable for the continuous development of the intelligent system they compose.

By incorporating testing strategies that support the evolution of software, intelligent systems can achieve higher levels of robustness and maintainability. This chapter reviews the concepts of traditional software automated testing. The main types of automated tests (Unit, Integration, Component Acceptance tests) are defined, the benefits of the testing practice are discussed, and Test Driven Development (TDD) is introduced.

3.1 Automated Tests

Automated tests are verification techniques that ensure software code works as expected. Automation applies software solutions to control test execution and obtain results with greater effi-

ciency and speed. As noted by Fowler (2018), automation allows for agile system validation, identifying security flaws, bugs, and other issues that could compromise the application. This process is essential in a context where code complexity and the frequency of changes are constant. The following sections determine the classification of automated tests, and the concept of mocks.

3.1.1 Classification of Automated Tests

Automated tests are often classified by the "Testing Pyramid", illustrated in Figure 3.1 which organizes categories based on complexity and recommended volume for effective coverage (Aniche, 2022).

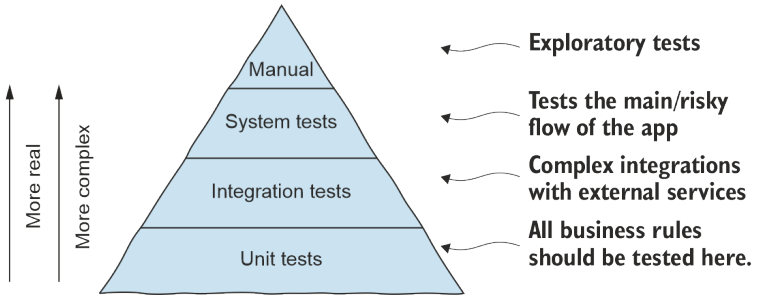

Figure 3.1: *Testing Pyramid: Classification of Automated Tests. Source: Aniche (2022)*

The following is a classification of automated tests:

- **Unit Tests:** Unit tests focus on testing individual units or components, typically the smallest testable parts of an application, such as functions or methods. These tests are essential for ensuring that each piece of code behaves as expected in isolation. Unit tests should:
  - Be small, fast, and independent from external systems (e.g., databases or network services) to provide rapid feedback. It is important to highlight that unit tests typically execute in milliseconds, offering a “fast safety net” that supports frequent code changes and continuous integration.
  - Target specific functionalities within the code, handling both expected and edge cases.
  - Run frequently to catch issues early, ideally integrated into a continuous integration (CI) pipeline for instant validation after each code change.

  Unit tests act as documentation for code behavior, helping developers understand the expected outputs and constraints of specific functions. Although they provide high code coverage, they lack the realism of integration with other components and may miss bugs that appear in combined workflows.

- **Integration Tests:** Integration tests verify that different modules or components of a system interact correctly with each other. They often include dependencies such as databases, third-party services, or file systems, which makes them slower and more complex than unit tests. Integration tests:
  - Focus on the interactions between modules to ensure correct data flows and dependency handling. These tests are particularly valuable for catching issues like data format mismatches and communication failures between components.

  - Should be carefully designed to cover critical interactions without duplicating unit test scopes, balancing test coverage with test complexity.
  - Reveal issues that unit tests may miss, such as unexpected behavior from external services or compatibility issues that arise only when components are tested together.

  Integration tests are essential for validating component compatibility and are generally executed less frequently than unit tests due to their added complexity and setup requirements.

- **System/Acceptance Tests:** System (or acceptance) tests validate the entire application as a whole, ensuring it functions as expected in an environment that closely resembles production. System tests:

  - Are high-level tests that simulate real-world scenarios, covering everything from data entry to output validation. It is important to see too the realism of these tests, which mimic actual user interactions.
  - Are essential for verifying end-to-end workflows and interactions across the entire application stack.
  - Require more resources and are prone to slower execution and occasional flakiness due to their complexity and full-system setup.

  These tests help uncover integration issues across the software stack and assess the system's behavior under various conditions, providing confidence in the application's readiness for production.

  Additionally, in the context of microservices, it is possible to divide system tests into two distinct categories: Component tests and End-to-End tests:

  - **Component Tests:** System tests that isolate a single service, while replacing all other services with *mocks*.
  - **End-to-End Tests:** System tests for the entire application.

  (Richardson, 2018)

- **Manual Tests:** At the top of the testing pyramid, manual tests represent the least automated layer and typically involve exploratory testing rather than scripted tests. Manual testing is valuable for assessing aspects that are difficult to automate, such as usability, visual layout, and unexpected user interactions. Manual tests:

  - Focus on the main or risky flows of the application that require human intuition or contextual evaluation, such as usability and design.
  - Are particularly useful for evaluating usability and design aspects that automated tests cannot cover.
  - Are time-consuming and resource-intensive, so they are performed less frequently compared to automated tests.

  Manual testing, although time-consuming, adds value by covering complex user interactions and exploratory testing, which are essential for final verification of the user experience.

3.1.2 Test Doubles

Test doubles are non-production ready, fake dependencies used in the context of automated testing in order to isolate a system from its dependencies (Khorikov, 2020).

They are commonly known as *mocks*, but there are up to five variations (Fowler, 2007):

- **Dummies**: objects with empty implementations.
- **Stubs**: objects that provide predefined responses for specific requests during a particular test.
- **Spies**: stubs that track information of interactions during the tests.
- **Mocks**: objects pre-programmed with expectations of calls they expect to receive during the test.
- **Fakes**: objects with non production-ready but working implementations, such as in memory databases.

3.2 Benefits of Automated Tests

Automated tests can be seen as the foundation of modern software development; they support the construction of an environment of continuous and iterative development that minimizes regressions. (Kon, 2008) By acting as a safety net, automated tests enable sustainable growth of a code base, especially for long-lived projects. Since the complexity of a software system tends to grow with each additional functionality, so does the chance of introducing bugs. Having a comprehensible suite of automated tests helps to avoid breaking existing functionality after each code change. (Khorikov, 2020)

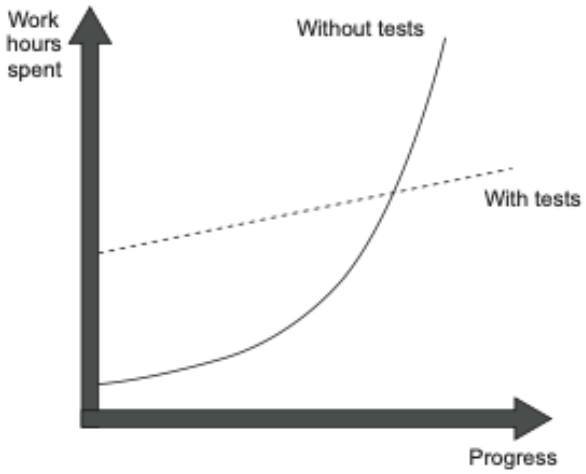

Figure 3.2: *Comparison of progress with and without tests. Adapted from Khorikov (2020).*

As illustrated in Figure 3.2, the use of tests significantly impacts the dynamic of growth in a software project by providing sustainability Khorikov (2020).

Additionally, the overall quality of the code is also improved by automated testing. As seen in Aniche (2015), many software developers believe that the design from their solutions is positively impacted by the practice of automated testing.

3.3 Test-Driven Development

Test-Driven Development (TDD) is a software development methodology in which automated tests drive the design and implementation of software. The primary objective of TDD is to reduce the uncertainty and fear often associated with developing complex systems. It emphasizes writing tests before the actual implementation code, which fosters a systematic, predictable approach to software creation.

The process is composed of a short, iterative cycle that encourages rapid feedback and continuous improvement. The typical steps in the TDD cycle are as follows:

1. **Write a Test**: Start by writing a concrete, runnable test that defines a specific behavior or scenario that the application should handle. This test is written before any functional code is developed, serving as a blueprint for the upcoming implementation.

2. **Make the Test Pass**: Write the minimum amount of code necessary to make the new test pass, while ensuring that all previously written tests also continue to pass. This step focuses on achieving the desired functionality with the simplest possible solution.

3. **Refactor**: Once the test passes, developers may refactor the code to improve its design and maintainability, making sure it remains clean, efficient, and scalable without changing its behavior.

4. **Repeat**: The cycle is repeated for each new piece of functionality. As new features are developed, new tests are written, and the system grows in small, incremental steps.

By following this cycle, TDD helps ensure that every change made to the system is tested, preventing regressions and promoting long-term maintainability. The automated tests suite act as a safety net, allowing developers to confidently refactor and extend the software without fear of breaking existing functionality. TDD also facilitates continuous integration, as the tests can be run frequently to ensure the software remains in a working state throughout its development. Ultimately, TDD encourages sustainable growth of software systems by emphasizing code quality and test coverage from the outset (Beck, 2002).

Chapter 4

SPIRA

This chapter describes the SPIRA project and dives deep into its latest version - SPIRA v2 -, made by Daniel Lawand, whose resulting design served as a reference for the work of this project.

4.1 The SPIRA Project

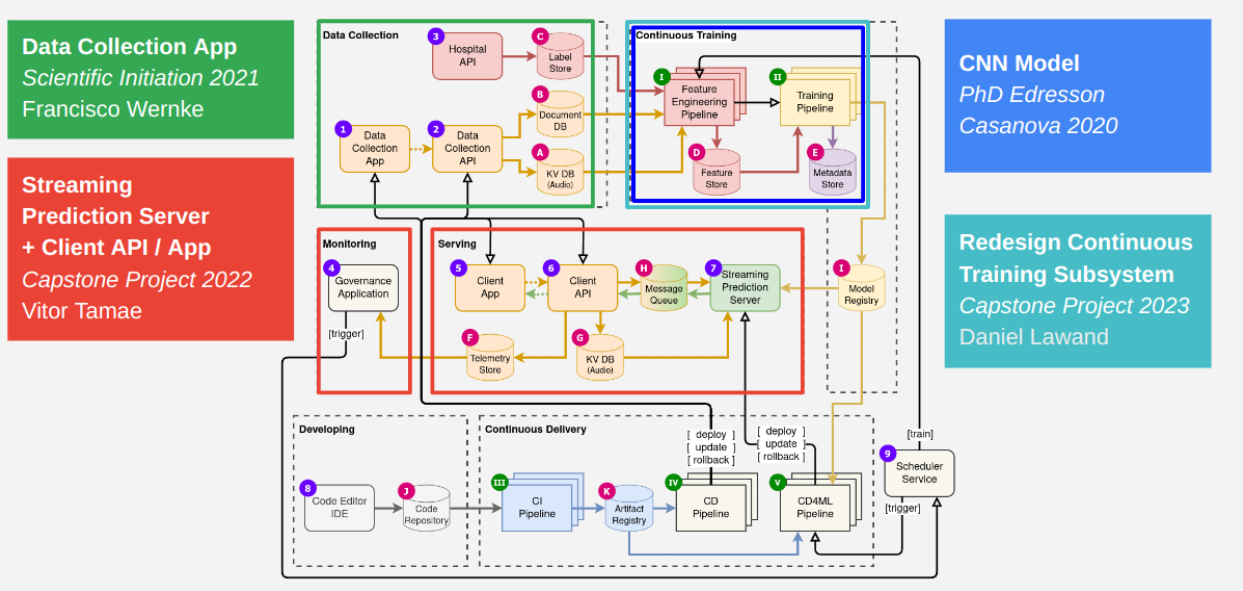

Figure 4.1: *The previous SPIRA distributed architecture illustration, containing information about components, their interactions, contributors, and year of development.*

SPIRA is a research project initiated during the COVID-19 pandemic with the aim of developing a system capable of performing pre-diagnosis of respiratory insufficiency, including the symptom of silent hypoxia. This system is based on speech analysis using Machine Learning models (Ferreira et al., 2022).

The system was built following a distributed architecture. Figure 4.1 illustrates the components and interactions, while the following section provides further details:

1. **Model trainer system**

   This system is responsible for training the Machine Learning models that analyze speech data. The process starts with gathering large amounts of audio data, which is then pre-processed and

labeled. Using CNNs, the Model Trainer extracts features from the audio signals and trains the model to recognize patterns indicative of respiratory issues. As more data is collected and incorporated into the system, the model's performance improves through retraining, making it more accurate over time.

2. **Model server and Inference API**

   Once the model is trained, the Model Server stores the final version of the trained model and provides access to it through an Inference API. This API allows medical professionals or other systems to submit new speech data and receive predictions. The API serves as a bridge between the trained model and the user, handling requests in real-time and providing pre-diagnosis results based on the model's analysis of speech data.

3. **Data collection API**

   This component is responsible for gathering the data needed to train the model. The Data Collection API works by collecting speech samples from patients, which are sent to a central database for storage and later processing. The data includes both normal and abnormal speech patterns, helping the model learn to distinguish between healthy individuals and those with potential respiratory insufficiency. The API handles all the back-end work of managing patient data, ensuring it's stored securely and made available for training.

(Ferreira et al., 2022)

Chapter 5

SPIRA V2

The SPIRA V2 architecture was designed with the goal of implementing a modular, maintainable structure for the SPIRA models training pipeline. Inspired by the Hexagonal Architecture pattern, this design aims to create a clear separation between core logic, adapters, and external dependencies. While SPIRA V2 has made strides toward this goal, there are areas where the architecture could be refined to fully embrace these principles. This chapter provides details about the current state of the Code Architecture, followed by the diagnosed opportunities for improvement.

5.1 Current Code Architecture

The current architecture was inspired by the **Hexagonal Architecture Pattern**. As such, the goal of the **Core package** is to concentrate the domain logic, while operations that depend on external services should be abstracted through **Ports** and implemented via **Adapters**. The main entry point for the application is located in the *pipeline* file, which is responsible for orchestrating and executing the model training pipeline.

Figure 5.2 illustrates the file structure of the current architecture, with further details provided in the following sections.

5.1.1 Core, Ports and Adapters Packages

In this version of the architecture, the Core package was responsible for all of the ML model's training logic, aggregating the training pipeline steps algorithms represented in figure 5.1.

In summary, the Core package's code is responsible for retrieving the data, applying feature engineering, formatting and structuring this data, executing the actual training, and storing the results. Since PyTorch was elected as the ML framework for the training algorithm, the data had to be conforming to the framework's format. To simplify this task, many models from the package were reused in SPIRA's Core, from basic data structures such as the Pytorch Tensor and Dataset, to more complex, ML training-specific components such as Pytorch Dataloaders, Schedulers, and Loss Calculators.

As for the Adapters package, the responsibilities elected were to perform adjacent, supportive operations, such as random numbers generation, path validation, and environment variables configuration.

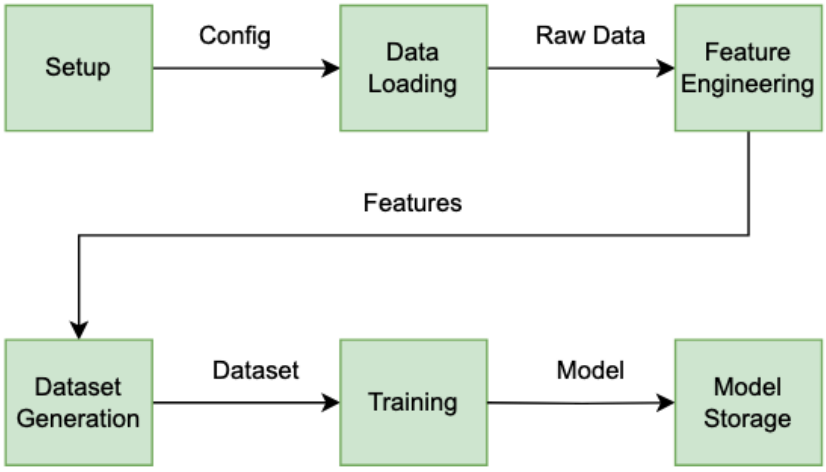

Figure 5.1: *Illustration for the machine learning model training pipeline steps. Adapted from Lawand (2023).*

Despite following the Hexagonal Architecture's principles, the creation of explicit Port interfaces was purposefully discarded, since it was not deemed necessary (Lawand, 2023).

5.1.2 The Pipeline file

The pipeline file serves as the central entry point of the application, functioning as the "main" file. Its primary responsibilities include configuring the application by reading environment variables, instantiating core services and adapters, and orchestrating the execution of the model training pipeline. It sequentially executes each step of the pipeline, manages the storage and forwarding of intermediate results between steps, and handles the final outcomes. This design centralizes the application's operational logic, making it a critical component of the architecture.

5.2 Opportunities for Enhancement

While SPIRA v2 has made strides toward the goal of productionizing the Model Trainer System, there are areas where the architecture could be refined. In particular, the analysis focused on implementing the Hexagonal Architecture by introducing key Ports and Adapters, clarifying the Core package's responsibilities, and enhancing service and pipeline structures.

5.2.1 Opportunity to Introduce Defined Ports and Adapters

In the Hexagonal Architecture, the core domain should be isolated from external dependencies, such as data sources, audio processing utilities, or model storage, interacting with them only through well-defined *ports*. However, the SPIRA V2 project currently lacks explicit *ports*, which are essential for creating a stable interface between the core domain and external systems. Without these *ports*, dependencies are often accessed directly from within the core, which reduces modularity and makes it more challenging to swap out or modify external systems without impacting the core. Implementing explicit port interfaces would facilitate code refactoring by clearly defining boundaries between the core business logic and adapters. With these defined *ports*, it could help reorganize the code-base to ensure that: adapters handle all interactions with external systems, and

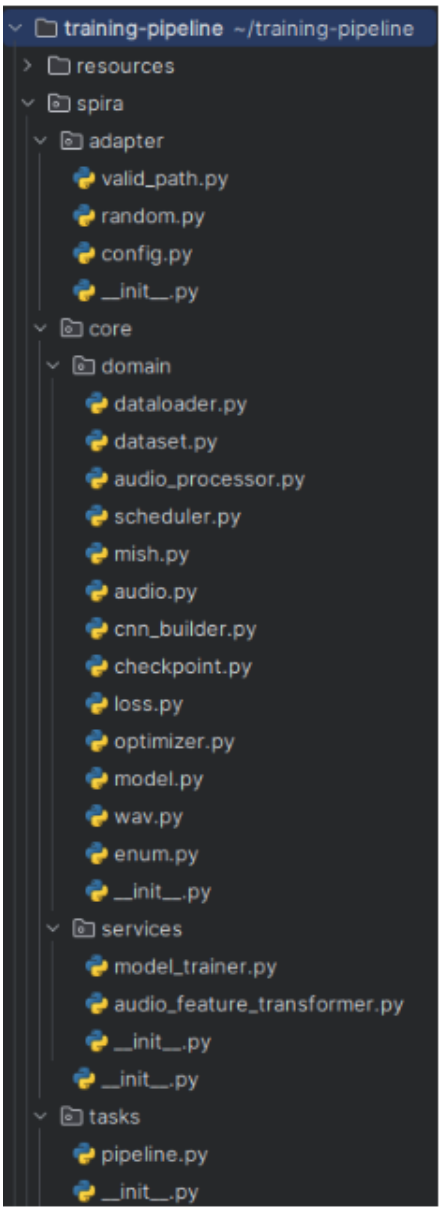

Figure 5.2: *Illustration for the file structure of the current architecture (V2). Source: Lawand (2023).*

the core focuses solely on business logic. This separation would improve maintainability, simplify testing, and make the architecture easier to evolve over time.

5.2.2 Clarifying Core Responsibilities

The `core` package includes files such as `audio.py`, `cnn_builder.py`, and `scheduler.py` that perform critical functions but also contain logic that could be separated for a cleaner architecture. Placing non-domain-specific logic—such as data loading or model configuration—into dedicated adapter modules would simplify the core and clarify its role. By doing so, the architecture would achieve a more distinct separation of concerns, allowing the core domain to concentrate on the essential training processes without being entangled with external systems.

Here are some examples:

- `model.py`: Core types like `FeaturesBatch`, `Parameter`, and `Model` are defined as dupli-

cates of types found in the PyTorch module, binding the core logic directly to PyTorch data structures.

- `checkpoint.py`: The checkpoint registration logic is coupled to the OS package for file system read and write operations, which complicates testing and portability.
- `scheduler.py`: The `Scheduler` class in the core is implemented as an extension of PyTorch's Scheduler class, further embedding PyTorch's dependencies within core logic.
- `audioprocessor.py`: The `AudioProcessor` class relies on torchaudio.transforms (a PyTorch-based library) for audio transformations, which ties the audio processing logic directly to PyTorch.

In a Hexagonal Architecture, these dependencies on frameworks and file systems would be abstracted behind *ports*, allowing them to be implemented as *adapters*. This approach would encapsulate the core domain, enabling it to interact with any external service or library through well-defined interfaces, making the system more adaptable, testable, and easy to maintain.

5.2.3 Enhancing Service and Pipeline Structure

The centralization of the application logic within the pipeline file raises several concerns. As the file executes each step sequentially, it lacks the ability to isolate individual pipeline steps. This limitation increases the fragility of the overall pipeline, as a failure in any step halts the entire execution, necessitating a complete restart from the beginning. Additionally, the interdependence of the steps prevents the pipeline from being paused and resumed remotely, which hinders integration with scheduler systems.

To address these limitations, the pipeline structure can be enhanced by decoupling individual steps and implementing mechanisms for state persistence. By saving the outputs of each step to external storage, the pipeline gains the ability to resume from the last completed step in case of a failure. This decoupling also enables the execution of isolated steps, improving flexibility.

Additionally, integrating a task orchestration framework, such as Apache Airflow, can provide advanced scheduling, retry capabilities, and failure isolation. With an orchestrator, each step in the pipeline can be managed as a separate task, executed sequentially or in parallel based on defined dependencies. This integration would also allow for better monitoring and logging, making the pipeline more robust and easier to maintain.

By adopting these enhancements, the Model Trainer can evolve into a modular and resilient system, capable of supporting complex workflows. Such improvements not only increase reliability but also enable integration with scheduling systems, promoting scalability and better alignment with modern software development practices.

Chapter 6

V3 Code Architecture

This chapter introduces the Code Architecture proposed for SPIRA's Model Trainer system. Referred to as the V3 Architecture, it adheres to the Hexagonal Architecture Pattern, abstracting framework-specific code behind ports and adapters to maintain a lean and flexible core package. The architecture was also designed to support integration with a Continuous Delivery pipeline and a Scheduling service. To achieve this, the system is organized into independent apps that communicate through well-defined interfaces and can be triggered by the Scheduling service. Although the system builds upon the redesign led by Daniel Lawand, it was completely rewritten with the incorporation of Automated Testing techniques to achieve a production-ready state.

The model trainer system code-base is organized into five main divisions:

1. **Core Package Code**: contains the domain-specific models and services that implement the core operations of the system.

2. **Ports/Adapters Package Code**: contains the interfaces (ports) used by the core package code to perform framework-specific operations or interact with external dependencies, as well as the adapter implementations for each port.

3. **Apps Code**: contains the main entry points for the model trainer system. Each app is responsible for setting up a specific step of the model trainer pipeline, including tasks such as reading environment variables, instantiating core services and their corresponding adapters, and executing the training step.

4. **Tests Code**: contains the code for automated testing, including test doubles, test setup and tear-down logic, dependency injection, and the test cases themselves.

5. **Infrastructure Code**: contains the code that handle the infrastructure for the model trainer system life-cycle, such as building Docker containers, configuring GitHub Actions for CI/CD pipelines, and managing dependencies.

Figure 6.1 provides a snapshot of the proposed file structure for the code-base.

The following sections present detailed descriptions and examples of code from each of these main divisions.

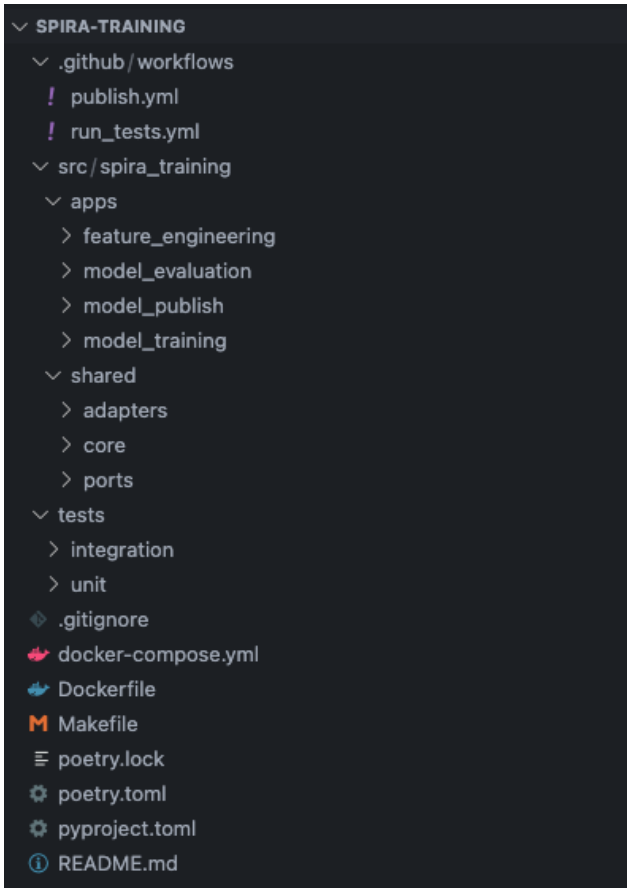

Figure 6.1: *Snapshot of the proposed architecture file structure.*

6.1 Core

The core package contains the domain modeling of the model trainer service. It defines the relevant application concepts as code in the *models* package and the system operations (pipeline steps) as classes in the *services* package. Figure 6.2 provides a snapshot of the source code files in each package.

The following is a list of some of the key concepts of the trainer system modeled in the core package:

1. **Audio**: The primary artifacts of the system, representing voice data recorded from patients. These audios are used for training and inference purposes and are modeled with sound wave representations and a hop length attribute.

2. **Dataset**: A batch of audios with predetermined labels—positive or negative—corresponding to the diagnosis of respiratory insufficiency for the patient who produced the audio.

3. **Trained Model**: The output of a model training process. A trained model performs inferences on audio data, can serialize its parameters, and is designed to load them to ensure reproducible performance.

At the current state of the project, two core services have been implemented: the Feature Engineering Service, and the Model Training Service.

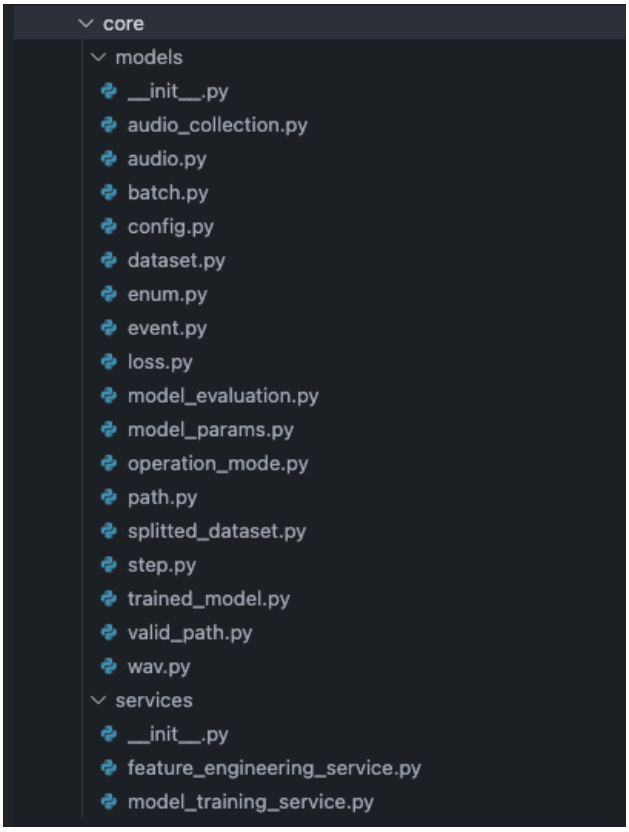

Figure 6.2: *Snapshot of the source code files in the core package.*

6.1.1 Feature Engineering Service

The Feature Engineering is a crucial component in the SPIRA model trainer system. It is responsible for multiple key tasks, including data loading, data injection, and transforming raw audio data into features that can be used for training models. By combining data loading and data injection with feature engineering, the app ensures a more efficient workflow and eliminates unnecessary redundancies in data handling.

```
1
2     ...
3
4     class FeatureEngineeringService:
5     def __init__(
6         self,
7         config: FeatureEngineeringConfig,
8         random_generator: RandomGenerator,
9         dataset_repository: DatasetRepository,
10        audios_repository: AudiosRepository,
11        file_reader: FileReader,
12        path_validator: PathValidator,
13        audio_processor_factory: Callable[[AudioProcessorConfig], AudioProcessor],
14        audio_feature_transformer_factory: AudioFeatureTransformerFactory,
15    ):
16        self.config = config
17        self.random_generator = random_generator
18        self.dataset_repository = dataset_repository
```

```
        self.audios_repository = audios_repository
        self.file_reader = file_reader
        self.path_validator = path_validator
        self.audio_processor_factory = audio_processor_factory
        self.audio_feature_transformer_factory = audio_feature_transformer_factory

    async def execute(self, save_dataset_path: Path) -> None:
        patients_inputs, controls_inputs, noises = self._load_data()
        patients_features, controls_features = self._process_audio_features(
            patients_inputs,
            controls_inputs,
            noises)
        dataset = self._generate_dataset(patients_features, controls_features)
        await self.dataset_repository.save_dataset(dataset, save_dataset_path)

```

Data Loading and Data Injection

The first step in the feature engineering process is **data loading**, which consists of reading the raw audio data from the specified storage location. This location can vary and may include sources such as local files, cloud storage, or databases. This step ensures that audio data is correctly retrieved, validated, and prepared for further processing.

The data loading workflow involves:

- **Reading Data Paths**: Extracting audio file paths from metadata repositories (e.g., CSV files), which organize paths for different datasets, such as patients, controls, and noise samples.
- **Validating Paths**: Ensuring the listed file paths are correct and accessible through validation mechanisms.
- **Fetching Audio Data**: Using these validated paths to retrieve the corresponding audio data for processing.

Once the data is loaded, the **data injection** process begins. This step feeds the audio data into the feature engineering pipeline, ensuring the data is distributed and available for subsequent transformations.

The reason data loading and data injection are integrated into the same service as feature engineering is to streamline the process. If each of these stages were separate services, it would introduce inefficiencies—specifically, we would need to first read the audio files during the data loading stage, store them in the audio storage, and then retrieve those same files again for feature engineering. By combining data loading, data injection, and feature engineering into one app, we avoid redundant I/O operations and reduce latency, ultimately enhancing the performance and efficiency of the system.

Feature Engineering

After data loading and injection, the raw audio data undergoes a series of transformations to extract meaningful features essential for training machine learning models. The **Feature Engineering** includes various feature transformers, such as:

- **MFCC (Mel-Frequency Cepstral Coefficients)**: Extracts coefficients that represent the short-term power spectrum of the audio signal.
- **Spectrogram**: Converts the audio signal into a visual representation of its frequency spectrum over time.
- **Melspectrogram**: Similar to a spectrogram but uses the Mel scale to represent frequencies, which aligns more closely with human perception of sound.

Each transformer is implemented as a separate class, with the main service class, FeatureEngineeringService, orchestrating the entire feature engineering process. This service is responsible for managing the data flow, applying necessary transformations, and ensuring that the resulting features are correctly formatted for model training.

6.1.2 Model Training Service

The Model Training Service contains the code responsible for producing and storing a build of a SPIRA trained model. Each step of the model training algorithm is mapped to one of the Ports, with corresponding Adapters injected into the Service to orchestrate the execution. The code for the service is illustrated in Figure 6.1.2.

```
1
2     class ModelTrainingService:
3         def __init__(
4             self,
5             dataset_repository: DatasetRepository,
6             dataset_splitter: DatasetSplitter,
7             model_trainer: ModelTrainer,
8             trained_models_repository: TrainedModelsRepository,
9         ):
10            self._dataset_repository = dataset_repository
11            self._model_trainer = model_trainer
12            self._trained_models_repository = trained_models_repository
13            self._dataset_splitter = dataset_splitter
14
15        async def execute(self, dataset_path: Path, trained_model_path: Path) -> None:
16            dataset = await self._dataset_repository.get_dataset(dataset_path)
17            splitted_dataset = self._dataset_splitter.split(dataset)
18            trained_model = self._model_trainer.train_model(
19                train_dataset=splitted_dataset.train_dataset,
20                test_dataset=splitted_dataset.test_dataset,
21                epochs=1,
22            )
23            await self._trained_models_repository.save_model(
24                trained_model, trained_model_path
25            )
26
```

The following is a listing of the steps and corresponding ports of the service:

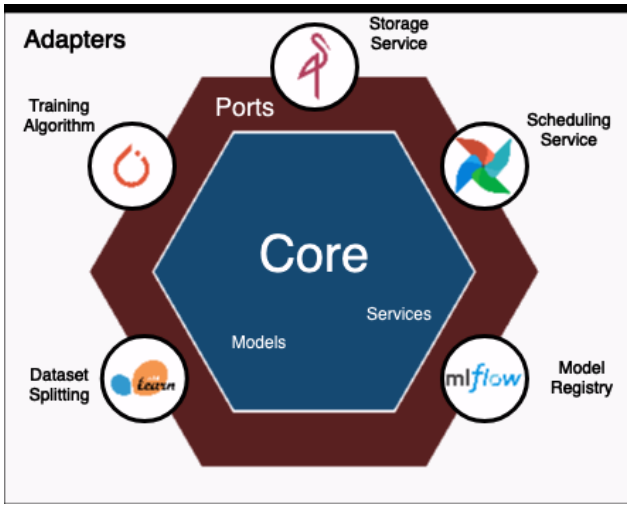

Figure 6.3: *Illustration for the hexagonal scheme of the current architecture. Displays some of the application's ports and corresponding adapters.*

1. **Dataset Splitting**: the partitioning of data provided by the Feature Engineering Service (describe in Section 6.1.1) into training and testing datasets. This step is executed by the Dataset Splitter Port (described in Section 6.2.1).

2. **Model training**: executing the model training algorithm. This step is executed by the Model Trainer Port (described in Section 6.2.2).

3. **Model storage**: storing the resulting trained model. This step is executed by a Repository Port (described in Section 6.2.4).

6.2 Ports and Adapters

In the V3 architecture, ports are interfaces accessed by the core package services to perform framework-specific operations or interact with external dependencies.

Figure 6.3 illustrates some of the application's ports and corresponding adapters, whose details are further explained in this section.

6.2.1 Dataset Splitter

This ports isolates the Core package from framework-specific details involved in the dataset splitting algorithm, such as dependency initialization, random state configuration, and data structures conversion.

Currently, the code base defines the Sci-kit Dataset Splitter Adapter. Having this algorithm abstracted as a port relieves the Core package from coupling to the Sci-kit library. By adhering to the Hexagonal Architecture pattern, it promotes extensibility and maintainibility.

6.2.2 Model trainer

This port isolates the Core package from framework-specific details involved in training a machine learning model, such as converting training and test data to tensors, performing loss calculations, and recalculating weights.

The primary goal of a Model Trainer Adapter is to receive the training datasets and hyperparameters, then output a trained model capable of:

- Performing inferences, and
- Serializing and loading its parameters to ensure reproducible performance.

The PyTorch Model Trainer serves as an Adapter for the Model Trainer Port. Its implementation was derived from Daniel Lawand's V2 Architecture and incorporates various substructures typical of ML algorithms.

Each of these substructures is modeled as an interface within the PyTorch Model Trainer package, which also includes their corresponding implementations. The following listing provides the definitions of some of these substructures:

- **Dataloader**: Responsible for organizing the dataset into batches for efficient processing. It abstracts the process of grouping data samples and labels into manageable units for training or inference.
- **Loss Calculator**: Handles the computation of the loss function, which measures the discrepancy between model predictions and ground-truth labels. It plays a critical role in guiding the optimization process by quantifying model performance.
- **Model**: Encapsulates the core logic of the machine learning model, including the ability to generate predictions and manage model parameters. It serves as the foundational structure for training and inference.
- **Optimizer**: Facilitates the optimization of model parameters during training. It abstracts the processes required for adjusting parameters to minimize the loss and supports mechanisms for managing the state of the optimization process.

Figure 6.4 illustrates Pytorch Model Trainer package's substructures interfaces and implementations folder structures.

Relevant to these substructures, there are also framework-specific model representations contained in the PyTorch package. The following listing provides definitions for some of these models:

- **Tensor**: The fundamental data structure in PyTorch, representing multi-dimensional arrays of numerical data. Tensors are the building blocks of the framework, used for both storing input data and intermediate computations during training and inference.
- **Label**: A scalar or vector associated with a data sample, representing the ground truth or target value. Labels guide the model during supervised learning by specifying the expected outputs.
- **Batch**: Represents a collection of data samples (features) and their corresponding labels, grouped together for efficient parallel processing during training or inference. In the system, batches are modeled as instances of the `PytorchBatch` class, which stores lists of features (`PytorchTensor`) and labels (`PytorchLabel`).

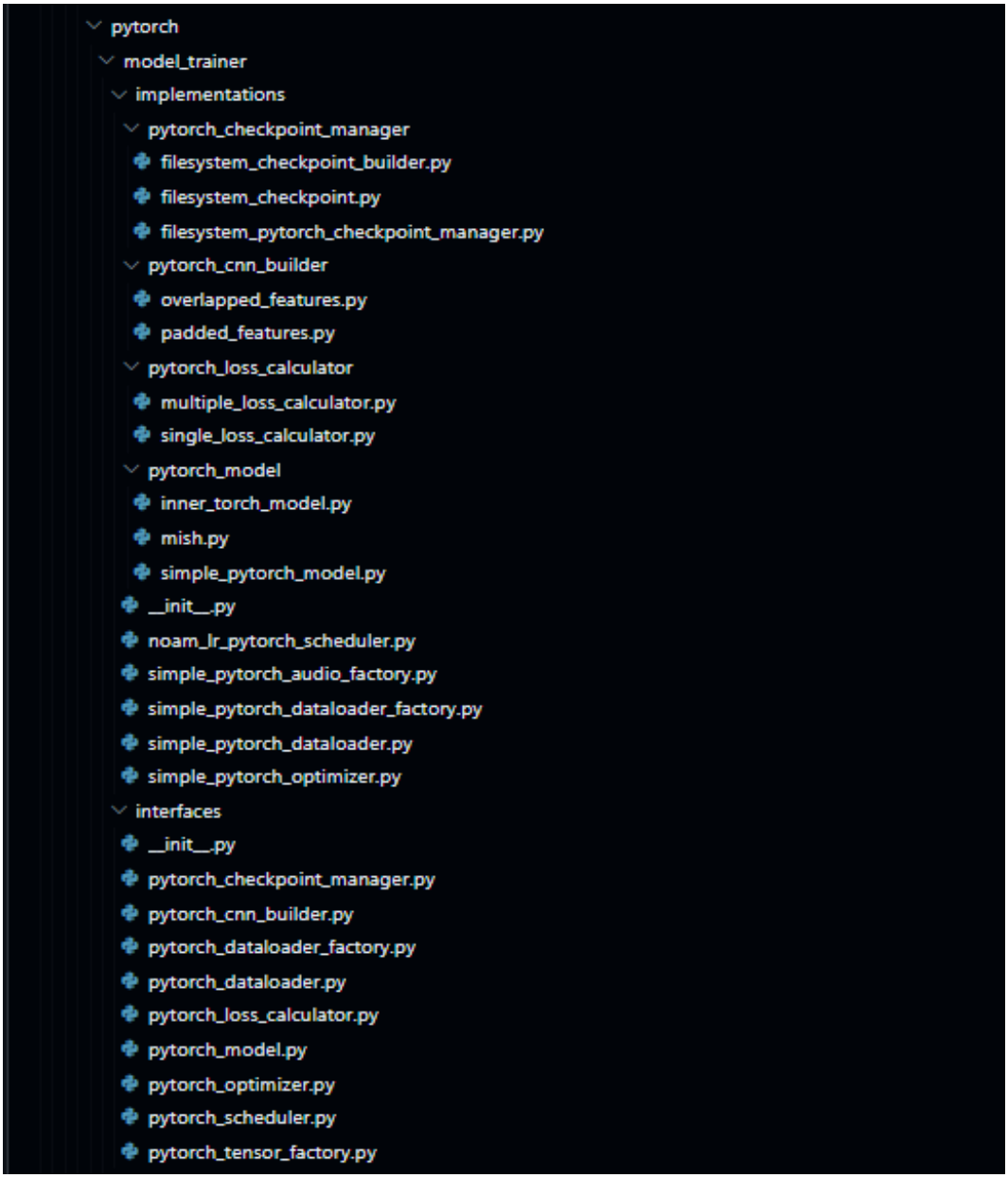

Figure 6.4: *Illustration for the Pytorch Model Trainer package's substructures interfaces and implementations folder structures.*

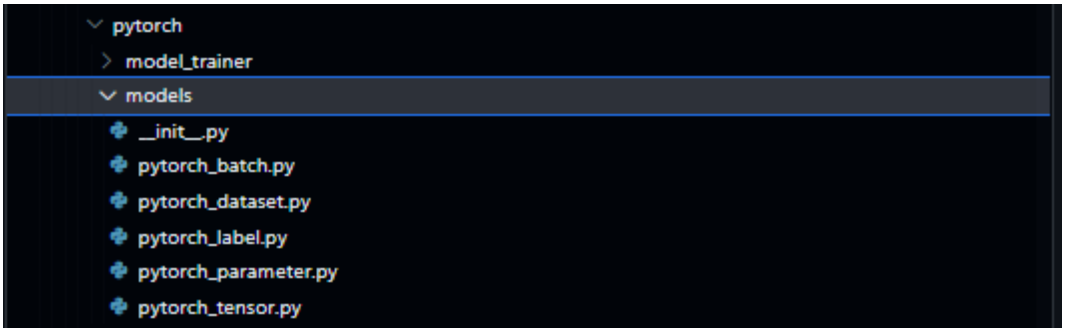

Figure 6.5: *Illustration for Pytorch package's models folder structure.*

- **Dataset**: Encapsulates a collection of features and labels, representing the complete data used for training or evaluation. The `PytorchDataset` class defines the dataset structure, enabling indexing (`__getitem__`) to retrieve feature-label pairs and providing a length method (`__len__`) to iterate through the dataset. The output of the dataset is typically fed into a Dataloader, which organizes data into batches.

Figure 6.5 illustrates Pytorch package's models folder structure.

6.2.3 Audio Feature Transformer

This port isolates the Core package from the specifics of transforming audio data into features. It defines the interface for applying various transformations to audio collections.

The primary goal of an **Audio Feature Transformer Adapter** is to receive raw audio data and output transformed features suitable for machine learning models. The following are key adapters for this port:

- **Padded Audio Feature Transformer:** Adds padding to audio collections to ensure consistent audio lengths. It uses an Audio Processor to handle the audio data and applies padding as needed.
- **Spectrogram Feature Transformer:** Transforms audio files into spectrograms using the `torchaudio.transforms` library.
- **MFCC Feature Transformer:** Converts audio files into Mel-Frequency Cepstral Coefficients (MFCC) features using the `torchaudio.transforms` library.
- **Audio Feature Transformer Pipeline:** Allows chaining multiple audio feature transformers together, applying each transformer in sequence to the audio collection.

6.2.4 Audios Repository

This port abstracts the retrieval of audio data from a specified path. It defines methods for accessing audio files, either as a list or individually.

The primary goal of an **Audios Repository Adapter** is to manage the storage and retrieval of audio data, ensuring that the Core package can access the necessary audio files without handling file system access details.

6.2.5 Config Loader

This port isolates the Core package from the specifics of loading configuration data. It defines the interface for reading configuration files and converting them into appropriate configuration objects.

The primary goal of a **Config Loader Adapter** is to provide a flexible mechanism for loading configuration data from various sources. The key adapter for this port is:

- **JsonConfigLoader:** Loads configuration data from JSON files, providing methods for loading general configuration data as well as specific feature engineering configurations.

6.3 Apps

A core objective of the redesign and development of the Model Trainer System was to support integration with a **Scheduling service** for the **Continuous Delivery** of trained models. To achieve this, the system was divided into **Apps**, designed to execute sequentially yet independently. These apps communicate through well-defined interfaces and can be triggered by the Scheduling service either:

1. **Manually**, via a User Interface,
2. **Programmatically**, in response to predefined events (e.g., acquisition of new training data).

Each app represents a single step in the model training pipeline. As the input for each step depends on the output of the previous step, the apps are designed to execute in sequence.

Each app serves as an **entry point** to the system, responsible for the setup and execution of a single pipeline step in the model training process. The main responsibilities of each app include:

1. **Input Handling**: Reading inputs through environment variables.
2. **Component Initialization**: Instantiating core services and adapters.
3. **Execution**: Running the corresponding core service to complete the pipeline step.

Below is a definition of the proposed apps and their corresponding inputs:

1. **Feature Engineering**:
   Purpose: This app is responsible for preprocessing the raw data into feature sets suitable for training.
   Input: Raw training data (CSV files, etc etc)
   Output: Processed features sets (dataset) saved in a storage bucket
2. **Model Training**:
   Purpose: This app handles training the machine learning model using the preprocessed dataset.
   Input: Feature sets (dataset) produced by the Feature Training app.
   Output: A trained model artifact (serialized model) saved in the model storage.

3. **Model Evaluation**:
   Purpose: This app evaluates the performance of the trained model against a predefined validation dataset, calculating important metrics.
   Input: The trained model artifact and a validation dataset.
   Output: Evaluation metrics stored in the database or log for review.

4. **Model Publish**:
   Purpose: This app publishes the trained model to the production environment, making it available for inference.
   Input: The trained model artifact and configurations settings for deployment.
   Output: A deployed model endpoint or a confirmation of deployment success.

6.4 Automated Tests

As one of the key aspects of a production-ready application, the development of a comprehensible automated test suite had a significant impact on the proposed architecture. Test Driven Development was applied to the Core package services code development, and several unit and integration tests were written for the Adapters package code. The resulting test suite is discussed in this section. Figure 6.6 presents the complete test suite in a test execution report.

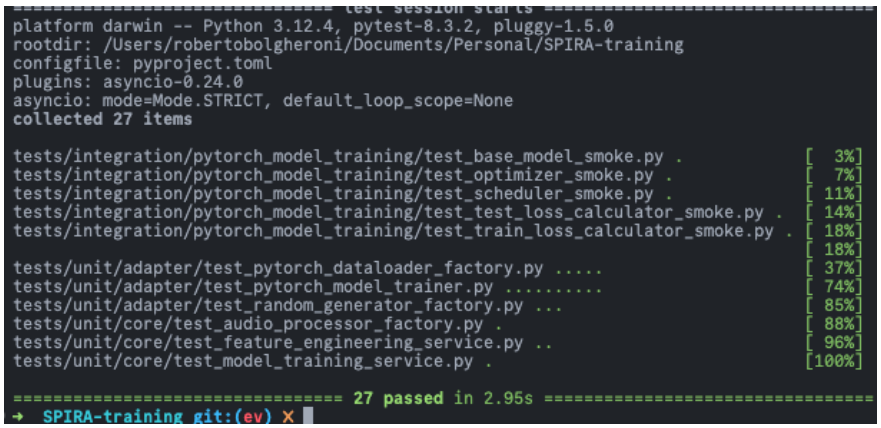

Figure 6.6: ***Tests Execution:*** *22 unit tests were added in 6 test files, covering the Core service classes and Adapters substructures; 5 smoke tests were added, covering all PyTorch Model Trainer substructures.*

6.4.1 Unit tests

In total, 22 unit tests were written in 6 test files, focusing on building a foundational test suite for the project. The services and the Pytorch Model Trainer adapter were re-developed applying TDD, with their design and sub-components defined prior to their implementation. Their unit tests focused on defining their interaction with sub-components and the expected outcomes of their execution. Figure 6.4.1 displays the test case for the Model Trainer Core service. This test focused on defining the final service execution by outlining the interaction with its sub-components.

```
1
2     ...
3
```

```
4     @pytest.mark.asyncio
5     async def test_execute():
6         # Arrange
7         dataset_path = Path("dataset_path")
8         trained_model_path = Path("any_model_storage_path")
9
10        base_dataset = make_dataset()
11        training_dataset = make_dataset()
12        test_dataset = make_dataset()
13        trained_model = make_trained_model()
14
15        dataset_repository = FakeDatasetRepository()
16        dataset_splitter = DatasetSplitterStub().with_split_result(
17            SplittedDataset(
18                train_dataset=training_dataset,
19                test_dataset=test_dataset
20            )
21        )
22        trained_models_repository = FakeTrainedModelsRepository()
23        model_trainer = FakeModelTrainer().with_train_result(trained_model)
24
25        await dataset_repository.save_dataset(
26            path=dataset_path,
27            dataset=base_dataset
28        )
29
30        sut = make_sut(
31            dataset_repository=dataset_repository,
32            dataset_splitter=dataset_splitter,
33            trained_models_repository=trained_models_repository,
34            model_trainer=model_trainer,
35        )
36
37        # Act
38        await sut.execute(
39            trained_model_path=trained_model_path,
40            dataset_path=dataset_path,
41        )
42
43        # Assert
44        assert model_trainer.called_with(
45            train_dataset=training_dataset, test_dataset=test_dataset
46        )
47        saved_model =
48            await trained_models_repository.get_model(
49                path=trained_model_path
50            )
51        assert saved_model == trained_model
52
```

Figure 6.4.1 displays the test cases for the *PytorchModelTrainer* adapter. By applying several TDD cycles these tests supported the progressive implementation of this component.

```
    ...

    def test_returns_trained_model():
        # Arrange
        setup = make_setup()
        sut = setup["sut"]

        # Act
        trained_model = sut.train_model(
            train_dataset=make_dataset(), test_dataset=make_dataset(), epochs=1
        )

        # Assert
        assert trained_model.dump_state() is not None

    def test_trains_with_each_batch_once():
        ...

    def test_trains_with_each_batch_for_each_epoch():
        ...

    def test_executes_optimizer_each_batch():
        ...

    def test_logs_loss_each_batch():
        ...

    def test_recalculates_weights_each_batch():
        ...

    def test_logs_test_loss_each_test_batch():
        ...

    def test_executes_scheduler_each_train_batch():
        ...

    def test_saves_checkpoint_each_test_batch():
        ...

    def test_saves_checkpoint_step():
        ...

```

In order to isolate components during unit tests, the practice of Test Doubles was adopted. In total, 18 fakes, mocks, and stubs were coded. Figure 6.4.1 displays the code for the *FakeModelTrainer* class, used at the Core Model Trainer Service test.

```
    ...

```

```
    class FakeModelTrainer(ModelTrainer):
        def __init__(self):
            self.called_with_args = None
            self.train_result: Optional[TrainedModel] = None

        def train_model(
            self,
            train_dataset: Dataset,
            test_dataset: Dataset,
            epochs: int
        ) -> TrainedModel:
            self.called_with_args = {
                "train_dataset": train_dataset,
                "test_dataset": test_dataset,
            }
            return self.train_result or make_trained_model()

        def called_with(self, train_dataset, test_dataset):
            return self.called_with_args == {
                "train_dataset": train_dataset,
                "test_dataset": test_dataset,
            }

        def with_train_result(self, trained_model: TrainedModel):
            self.train_result = trained_model
            return self

```

6.4.2 Integration Tests

Due to the absence of concrete test data, the project's integration tests were written in the form of Smoke Tests for the Pytorch Model Trainer adapter. These tests were written after implementation, and they ensure that the sub-components for the Pytorch Model Trainer adapter could be properly initialized and executed. In total, 5 test files were added for these sub-components. Additionally, a test configuration file was added for initializing each sub-component.

Figure 6.4.2 displays the code from the initialization and the smoke test for the *PytorchLossCalculator* class.

```

    ...

    @pytest.fixture()
    def train_loss_calculator(
        single_train_loss_calculator: SingleLossCalculator,
    ) -> PytorchLossCalculator:
        return AverageMultipleLossCalculator(
            single_loss_calculator=single_train_loss_calculator,
        )

    ...
```

```
13
14      def test_train_loss_calculator_smoke(
15          train_loss_calculator: PytorchLossCalculator
16      ):
17          try:
18              train_loss_calculator.calculate_loss(
19                  predictions=torch.tensor([1.0]), labels=torch.tensor([1.0])
20              )
21              train_loss_calculator.recalculate_weights()
22          except Exception as e:
23              assert False, f"Failed with exception: {e}"
24
25          assert True
26
27
```

6.4.3 Tests for ML

With the project's goal of developing a comprehensive automated test suite, the approach was to create a foundational test coverage. Due to the focus on adding simple unit and integration tests, no machine learning-specific tests were added, such as data validation, model quality and bias validation, or data transformation validation.

6.5 Infrastructure

The infrastructure for SPIRA V3 was designed to ensure the system is robust, maintainable, and aligned with modern software engineering practices. This section describes the components and processes implemented to support the lifecycle of the model trainer system, enabling seamless development, testing, and deployment.

6.5.1 Docker

Docker plays a central role in SPIRA V3's infrastructure by containerizing all system components. Each application and its dependencies are encapsulated within a Docker image, ensuring consistency across environments. This approach minimizes discrepancies between development, testing, and production setups, streamlining the deployment process.

Key enhancements in Docker usage for SPIRA V3 include:

- Creating optimized Dockerfiles for the model trainer components, reducing build times and improving portability.
- Integrating automated testing into the Docker setup, allowing unit and integration tests to be executed consistently within containerized environments.
- Ensuring all containers are lightweight and reusable, following best practices for dependency management and image layering.

These steps ensure that the entire system remains predictable and easy to deploy in diverse scenarios.

6.5.2 GitHub Actions Pipelines

A robust Continuous Integration/Continuous Deployment (CI/CD) pipeline was implemented using GitHub Actions to automate the development life-cycle. This pipeline integrates various stages of validation and deployment, ensuring code quality and reducing manual effort. The key steps of the pipeline include:

1. **Code Validation:** Linting, type checking, and running unit tests are triggered automatically upon every code push or pull request. This ensures that all contributions meet quality standards.

2. **Integration Testing:** Smoke tests and integration tests are executed within the pipeline to validate the interactions between different components of the system.

3. **Artifact Management:** Docker images are built and pushed to the GitHub Container Registry, ensuring that validated and ready-to-use images are always available for deployment.

4. **Deployment Steps:** Automated deployment scripts deploy validated images to target environments, providing a seamless and reliable release process.

Figure 6.7 is a diagram showing the flow of github actions:

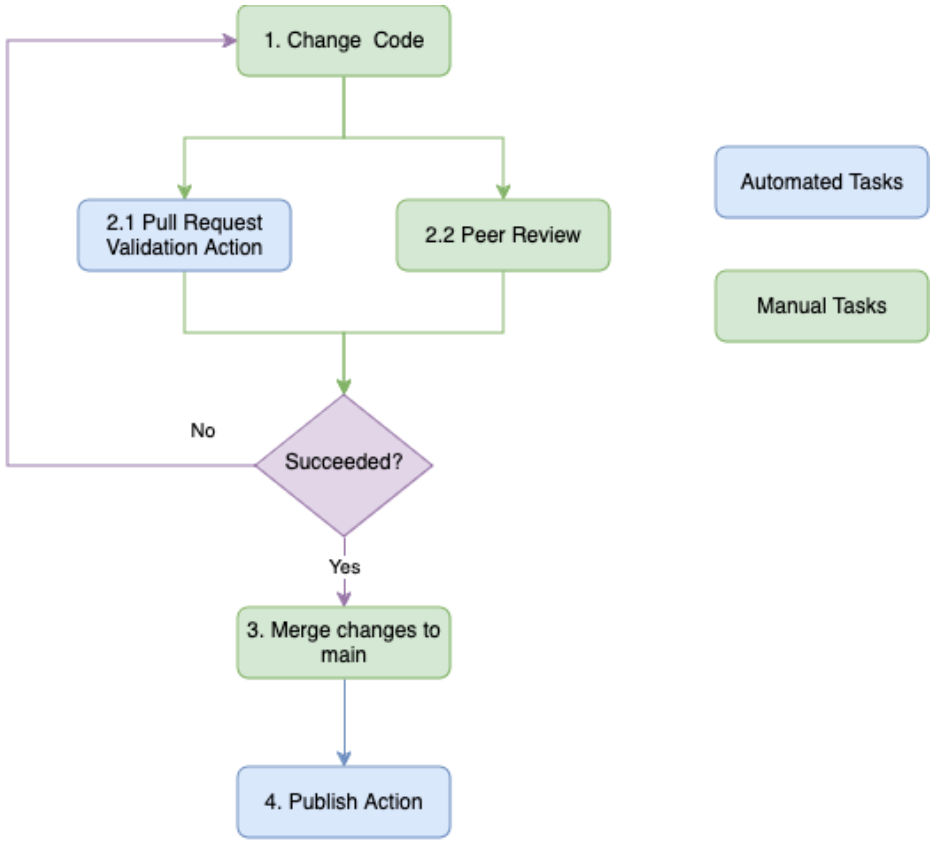

Figure 6.7: *Flow of GitHub Actions in a CI/CD Pipeline: The diagram outlines the process from code changes to deployment, including automated validation, manual peer review, merging to main, and production deployment.*

The pipeline comprises the following steps:

1. **Change Code:** Developers push changes to the repository, triggering the CI/CD pipeline.
2. **Validation and Review:** This stage is divided into two sub-tasks:
   (a) **Pull Request Validation Action:** Automated tasks such as linting, type checking, unit tests, and integration tests are executed.
   (b) **Peer Review:** The pull request undergoes manual review by team members to ensure code quality and adherence to best practices.
3. **Merge Changes to Main:** Upon successful validation and review, the changes are merged into the main branch.
4. **Publish Action:** The pipeline deploys the validated code to the production environment or other specified deployment targets.

The flow is structured to ensure that only validated and peer-reviewed code reaches production. If any step fails, the process reverts to the initial stage for corrections. This ensures a rigorous validation process while maintaining an efficient development workflow.

The CI/CD pipeline not only increases development efficiency but also aligns with the principles of Continuous Delivery, as discussed in Chapter **??**.

6.5.3 Tests and Type Checking

Testing and validation form a cornerstone of SPIRA V3's infrastructure. Key measures implemented include:

- **Unit Tests:** Comprehensive tests validate the functionality of individual components, ensuring that changes do not introduce regressions.
- **Integration Tests:** Tests validate the interaction between system components, focusing on data flows and dependency compatibility.
- **Type Checking:** The `mypy` tool was integrated into the pipeline to enforce type consistency, catching potential errors before runtime.

These practices ensure that SPIRA V3 maintains high code quality, stability, and readiness for deployment.

6.5.4 Summary of Infrastructure Design

The infrastructure developed for SPIRA V3 reflects a commitment to scalability, maintainability, and reliability. By leveraging Docker for containerization, implementing a comprehensive CI/CD pipeline with GitHub Actions, and incorporating rigorous testing and type-checking measures, the system is well-prepared for continuous improvement and deployment in diverse environments.

Chapter 7

Results

This chapter summarizes the improvements achieved during this project with respect to SPIRA's V2 architecture. The three main axes of changes were: Code Architecture, Infrastructure and Automated Testing.

Figure 7.1 presents the updated SPIRA architecture, showcasing the key improvements introduced in this project.

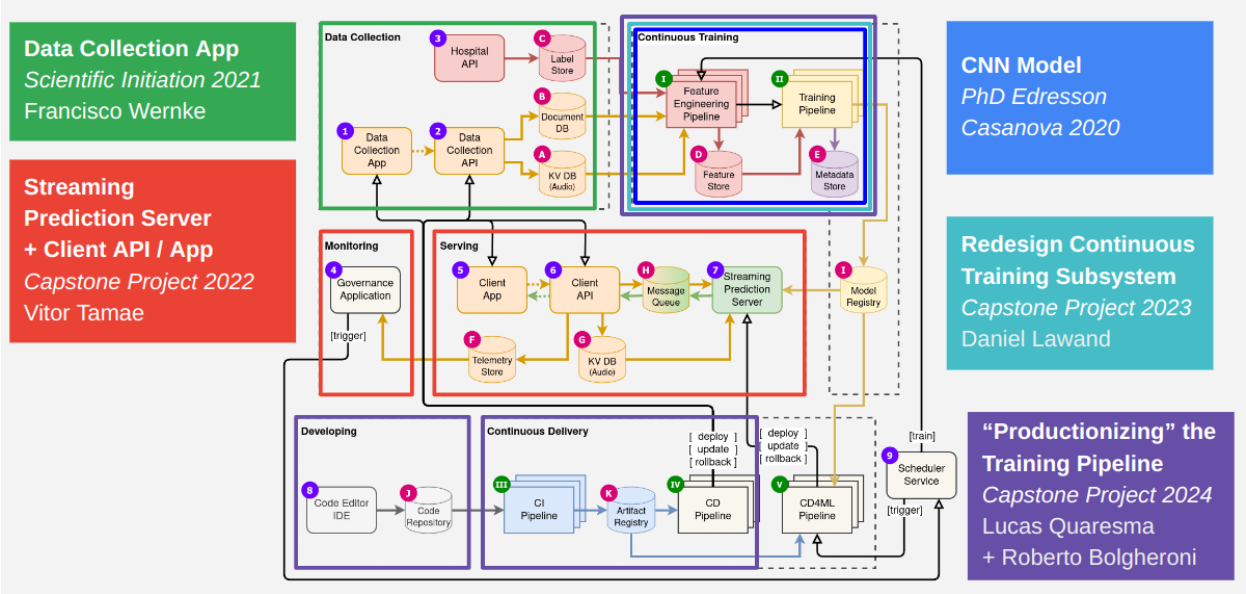

Figure 7.1: *The current SPIRA distributed architecture, containing information about components, their interactions, contributors, and year of development.*

7.1 Code Architecture

This section describes the improvements made by this project to the code architecture of the SPIRA model trainer system's V2 design. With a focus on developing a continuous delivery pipeline and integrating it to the remaining microservices, the training algorithm was separated into separate sub-services, named apps, with well defined interfaces. Most notably, the hexagonal architecture started in the V2 design was expanded, abstracting more dependencies behind ports, and creating service classes for each app. In addition, the goal of improving the overall clarity and readability of complex algorithms was also tackled during the project.

7.1.1 The Apps division

The transition from SPIRA V2 to SPIRA V3 introduced significant improvements to the organization and modularity of the model trainer pipeline through the concept of "apps." This section compares how the pipeline was structured in SPIRA V2, as described in Chapter 5, with the enhancements introduced in SPIRA V3, detailed in Chapter 6.

In SPIRA V2, the entire model trainer pipeline was orchestrated through a single pipeline file. This central script was responsible for configuring the environment, initializing services, and executing all stages of the pipeline sequentially. While this approach provided a clear entry point for the application, it presented several challenges:

- **Monolithic Structure:** The tightly coupled design made it difficult to isolate, test, or modify individual steps in the pipeline.
- **Lack of Modularity:** All stages were implemented within the same context, limiting the ability to reuse or reorganize components for different use cases.
- **Limited Fault Tolerance:** Any failure in a single step would halt the entire pipeline, requiring a complete restart.

To address these limitations, SPIRA V3 introduced the concept of dividing the pipeline into distinct, modular apps, each with well-defined responsibilities. As detailed in Chapter 6, the new design organizes the pipeline into four independent apps:

1. **Feature Engineering App:** Preprocesses raw audio data into feature sets suitable for training.
2. **Model Training App:** Trains machine learning models using the processed datasets.
3. **Model Evaluation App:** Evaluates the performance of trained models against a validation dataset.
4. **Model Publish App:** Deploys the trained model to the production environment.

Key advantages of this new approach include:

- **Improved Modularity:** Each app is self-contained, allowing independent development, testing, and deployment.
- **Enhanced Fault Tolerance:** Outputs from each app are stored in external storage, enabling the pipeline to resume from the last completed step in the event of a failure.
- **Integration with Scheduling Services:** The apps are designed to integrate seamlessly with task schedulers, such as Apache Airflow, allowing for advanced orchestration and monitoring capabilities.

Figure 7.2 illustrates the training pipeline steps and their standing in the apps division.

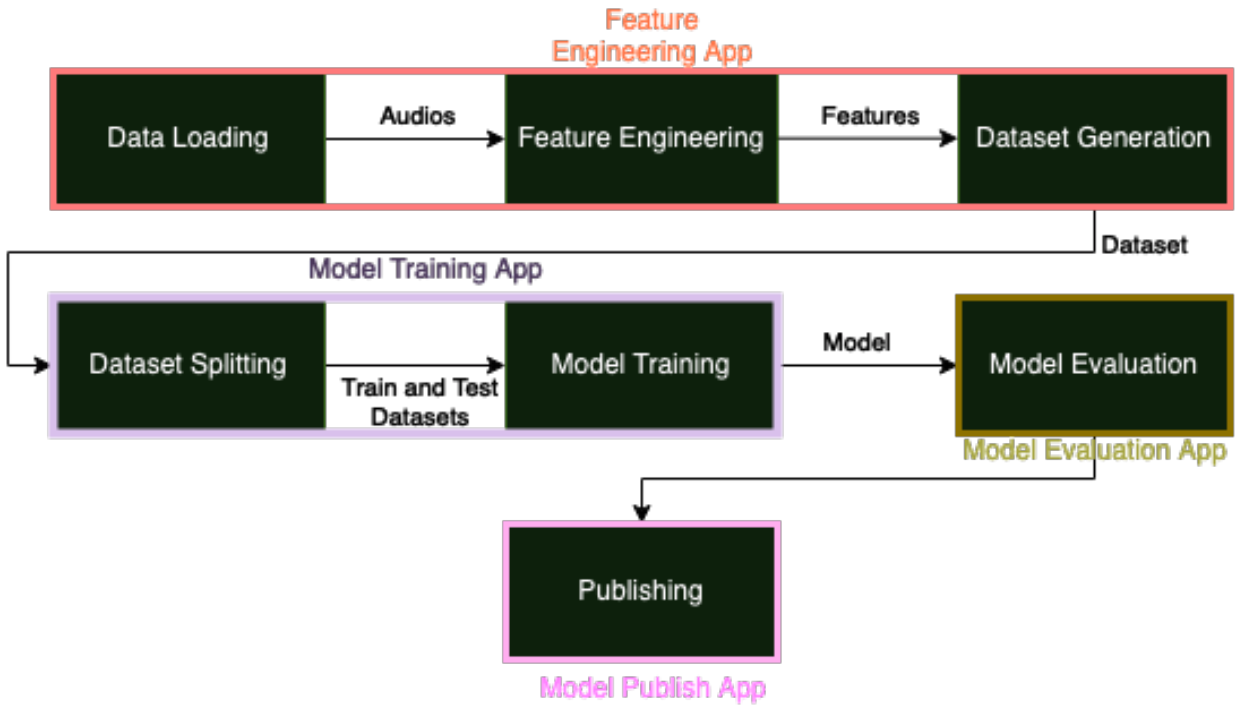

Figure 7.2: *The model training pipeline steps were reorganized into four independent applications, balancing modularity and efficiency by grouping related steps where separation would be impractical.*
Note: *The* ***Model Evaluation*** *and the* ***Model Publish*** *apps were not part of the original v2 architecture. They were designed and structured as part of future work in the SPIRA roadmap*

7.1.2 Expanding the Hexagonal Architecture

The adoption of Hexagonal Architecture principles significantly enhanced the solution's adaptability to changes in external systems and technologies. New data sources, external APIs, and ML frameworks can now be integrated seamlessly by creating new Adapters, without altering the core domain logic. This separation of concerns not only improves maintainability but also enables parallel development of system components and fosters collaboration within the SPIRA project.

Figure 7.3 compares the V2 and V3 designs, highlighting the architectural evolution. Key components, such as Repositories for data storage, the Model Trainer port for ML framework-specific logic, and the File Reader port for CSV file processing, were abstracted through Ports. These abstractions enhanced modularity and decoupled core logic from implementation details, laying a strong foundation for future scalability and flexibility.

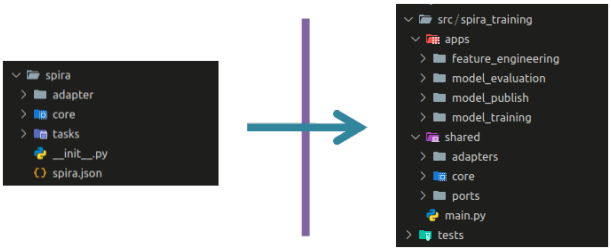

Figure 7.3: *Comparison of SPIRA V2 and SPIRA V3 architectures. SPIRA V2 (left) had a monolithic structure, while SPIRA V3 (right) introduces modularization with distinct domains and ports and adapters, improving maintainability and scalability*

7.1.3 Opportunities for Enhancement

The transition from SPIRA V2 to SPIRA V3 involved addressing key opportunities for enhancement identified in Chapter 5. These opportunities were focused on improving modularity, maintainability, and scalability. This section discusses how these challenges were tackled in the updated architecture presented in Chapter 6.

Introducing Defined Ports and Adapters

As outlined in Section 5.2.1 of Chapter 5, SPIRA V2 lacked explicit port definitions, leading to a tighter coupling between the core domain logic and external dependencies. This design limited flexibility and made it difficult to replace or update external components without impacting the core logic.

In SPIRA V3, explicit ports were introduced to adhere to the principles of Hexagonal Architecture, as detailed in Chapter 6. These ports define clear interfaces between the core domain and external dependencies. Key improvements include:

- **Dataset Splitter Port:** Abstracts the logic for dataset splitting, allowing different implementations to be used without modifying the core.
- **Model Trainer Port:** Isolates the model training logic from specific frameworks like PyTorch, enhancing adaptability.
- **Audio Feature Transformer Port:** Decouples audio preprocessing from the core logic, enabling flexibility in implementing feature extraction techniques.

These changes improved the modularity and extensibility of SPIRA V3, making it easier to integrate new technologies or adapt to changing requirements.

Clarifying Core Responsibilities

In SPIRA V2, as noted in Chapter 5, the core package included logic that overlapped with external dependencies, such as PyTorch-specific implementations and file system operations. This lack of separation of concerns made the code harder to maintain and test.

SPIRA V3 addressed this by refactoring the core package to focus solely on domain-specific logic, as described in Chapter 6. Key actions taken include:

- Moving framework-specific logic, such as data loaders and schedulers, to adapters.
- Isolating file system operations into dedicated repositories, ensuring that the core logic remains framework-agnostic.
- Consolidating core services to encapsulate business logic, such as feature engineering and model training, without reliance on external libraries.

These adjustments made the code-base more modular and easier to maintain, aligning it with best practices in software engineering.

Enhancing Service and Pipeline Structure

SPIRA V2's pipeline file, as discussed before, centralized the execution of the entire model training process. While functional, this design limited fault tolerance and prevented the pipeline from being paused or resumed.

SPIRA V3 restructured the pipeline into independent, modular services. This new approach allows us to introduce:

- **State Persistence:** Intermediate outputs from each pipeline step are saved, allowing the process to resume from the last completed step in case of failure.
- **Decoupled Services:** Each step of the pipeline is implemented as an independent app, facilitating easier testing, scaling, and modification.
- **Task Orchestration:** Integration with schedulers like Apache Airflow enables advanced features such as parallel execution, retry mechanisms, and detailed monitoring.

This redesign significantly improved the pipeline's resilience, flexibility, and maintainability, making it better suited for production environments.

7.2 Infrastructure

This section describes the achieved results of the project in the building of the infrastructure design of SPIRA V2 and V3, and the Continuous Delivery pipeline for SPIRA's ML models. The following steps will be described: the creation of type checking and automated testing validation steps, the creation of the GitHub Actions pipeline, and the integration of the pipeline with the GitHub container registry for publishing of the latest versions of the model trainer service.

7.2.1 Model trainer validation steps

In SPIRA V2, the infrastructure was limited to basic containerization with Docker. While this provided consistency across environments, it lacked critical validation mechanisms such as automated testing, type checking, and structured code validation, which are essential for a production-ready system.

SPIRA V3 introduced a robust validation process to address these gaps. Key improvements include:

- **Type Checking:** Scripts using tools like `mypy` were added to enforce type safety and minimize runtime errors, significantly improving code reliability.
- **Static Code Analysis:** Tools like `flake8` or `pylint` were integrated to ensure adherence to coding standards and detect potential issues early in the development cycle.
- **Automated Testing:** Docker images were enhanced to include automated unit and integration tests (Chapter 6.4), ensuring that functionality could be validated consistently across all stages of development.

These improvements add an essential layer of robustness to the development pipeline, aligning with the continuous delivery objectives described in Chapter 2.3.

7.2.2 GitHub Actions pipeline

Another of the limitations of SPIRA V2 was the absence of a Continuous Integration/Continuous Deployment (CI/CD) pipeline. So, to complement the enhancements to validation, SPIRA V3 implemented a GitHub Actions-based CI/CD pipeline as described in **??**. This pipeline introduces automation to key stages of the development life-cycle:

1. **Code Validation:** Every code change triggers the pipeline, automatically running tasks such as type checking, linting, and executing unit tests.
2. **Integration Testing:** Smoke and integration tests are executed to validate interactions between components, ensuring system stability.
3. **Container Registry Integration:** A GitHub Container Registry was created and integrated with the pipeline, allowing for automated building, publishing, and storage of Docker images.
4. **Deployment Automation:** Validated images are seamlessly deployed to the designated environment, facilitating a smooth and consistent release process.

These additions streamline the development workflow and provide a solid foundation for ongoing enhancements.

7.3 Automated Testing

Building a comprehensive automated testing suite for SPIRA's Model Trainer System was one of the key objectives of this project. The previous version of the system (described in chapter 5), did not have automated testing coverage, which hindered the implementation of a CI/CD pipeline. This proposed version, as detailed in Section 6.4, successfully produced a comprehensive testing suite for the system.

Unit tests were added for the Core services and the Pytorch Model Trainer adapter. By applying TDD, new abstractions were developed, enhancing the overall design of the solution through improved decoupling, maintainability, and extensibility.

Additionally, smoke integration tests were implemented for complex machine learning components. These tests provide critical coverage and support the system's continuous evolution as new data becomes available.

The resulting test suite forms the foundation for Continuous Delivery and iterative development within the SPIRA project. Future enhancements to the code base will be supported by automated testing, promoting faster feedback cycles and faster evolution of the project.

7.4 Summary

The results of this project demonstrate significant advancements in the design, infrastructure, and testing capabilities of SPIRA's model trainer system. By transitioning from SPIRA V2 to V3, the system has evolved into a more modular, maintainable, and scalable solution, better suited for continuous delivery and production environments. These improvements represent a substantial step forward in the system's maturity, laying a robust foundation for future enhancements. As a result,

SPIRA is now better equipped to adapt to evolving challenges while maintaining high standards of reliability and scalability.

Chapter 8

Conclusion

The primary objective of this project was to productionize SPIRA's model trainer system by completing the continuous delivery pipeline. This effort focused on making the training and deployment of new models reproducible, safer, and faster through automation, ultimately enabling shorter development cycles, enhanced maintainability, and improved software quality.

Key advancements were achieved during the project. A foundational automated test suite was developed, enabling faster feedback cycles, regression prevention, and improved code quality to support future iterative development. The training pipeline was restructured into independent, modular applications, significantly improving its resilience, flexibility, and maintainability. This redesign also paves the way for integration with a scheduling service, a critical step toward fully realizing the continuous delivery pipeline.

While significant progress has been made, two main objectives remain for future work: integrating the system with a scheduling service and implementing automated tests focused on machine learning-specific aspects. These enhancements represent opportunities to further optimize the SPIRA model trainer system and solidify its position as a robust, production-ready solution.

Bibliography

AWS Docs Amazon. Training ML Models, 2024. URL https://docs.aws.amazon.com/machine-learning/latest/dg/training-ml-models.html. 4

Mauricio Aniche. **Effective Software Testing: A developer's guide**. Manning Publications, 2022. ISBN 9781633439931. 8

Maurício Aniche. Does test-driven development improve class design? A qualitative study on developers' perceptions, 2015. URL https://www.ime.usp.br/~gerosa/papers/Aniche-Gerosa2015_Article_DoesTest-drivenDevelopmentImpr.pdf. 10

Kent Beck. **Test Driven Development: By Example**. Addison-Wesley Professional, 2002. ISBN 9780321146533. 11

Edresson Casanova, Lucas Gris, Augusto Camargo, Daniel da Silva, Murilo Gazzola, Ester Sabino, Anna S. Levin, Arnaldo Candido, Sandra Aluisio, and Marcelo Finger. Deep learning against covid-19: Respiratory insufficiency detection in brazilian portuguese speech. 2021. doi: 10.18653/v1/2021.findings-acl.55. 3

Alistair Cockburn. Hexagonal architecture, 2008. URL https://alistair.cockburn.us/hexagonal-architecture. 6

Renato Ferreira, Dayanne Gomes, Vitor Tamae, Francisco Wernke, and Alfredo Goldman. Spira: Building an intelligent system for respiratory insufficiency detection. In **Anais do II Workshop Brasileiro de Engenharia de Software Inteligente**, pages 19–22, Porto Alegre, RS, Brasil, 2022. SBC. doi: 10.5753/ise.2022.227048. URL https://sol.sbc.org.br/index.php/ise/article/view/22530. 12, 13

Martin Fowler. Mocks aren't stubs, 2007. URL https://martinfowler.com/articles/mocksArentStubs.html#TheDifferenceBetweenMocksAndStubs. 10

Geoff Hulten. **Building Intelligent Systems: A Guide to Machine Learning Engineering**. Apress, 2018. ISBN 978-1484234310. 4

Vladimir Khorikov. **Unit Testing Principles, Practices, and Patterns**. Manning Publications, 2020. ISBN 9781617296277. 10

Fábio Kon. A Importância dos Testes Automatizados, 2008. URL https://www.ime.usp.br/~kon/papers/EngSoftMagazine-IntroducaoTestes.pdf. 10

Daniel Lawand. Enabling MLOps in the SPIRA Training Pipeline, 2023. URL https://github.com/danlawand/MAC0499/blob/main/docs/monograph.pdf. 3, 15, 16

MarkovML. Model Deployment, 2023. URL https://www.markovml.com/blog/model-deployment. 4

Chris Richardson. **Microservices Patterns: With examples in Java**. Manning Publications, 2018. ISBN 9781617294549. 9

Danilo Sato, Arif Wider, and Christoph Windheuser. Continuous delivery for machine learning. **Martin Fowler**, 2019. 3, 5, 7

Ben Wilson. **Machine Learning Engineering in Action**. Manning, 2022. ISBN 978-1617298714. 5