Building an intelligent system to detect respiratory insufficiency

Vitor Daisuke Tamae

Capstone Project
Presented to the discipline
MAC0499

Supervisors:
Prof. Dr. Alfredo Goldman
Prof. Dr. Marcelo Finger
MSc. Renato Cordeiro Ferreira

São Paulo, December of 2022

Contents

https://pt.overleaf.com/project/63014d167957d86e6cf4ada5

Chapter 1

Introduction

Respiratory insufficiency or failure is a well-known medical symptom resultant from an inadequate gas exchange performed by the respiratory system (Cam65). It may cause severe symptoms such as hypoxemia, a reduction in the concentration of oxygen in bloodstream, and hypercapnia, a rise in arterial carbon dioxide levels (RK03). These conditions may lead to cough, tiredness, shortness of breath, and in extreme cases, death. This symptom may be caused by various diseases. Examples of them are the flu, severe asthma and heart conditions (SLK09; CCN$^+$00; DS94).

In 2020, respiratory insufficiency was identified as one of the severe symptoms of COVID-19 infection (SVdB22). The symptom is aggravated by a condition called *silent hypoxia*, low blood oxygen concentration without breath shortness (CF20)(TLJ20), which particularly hinders its diagnosis.

There were several attempts to obtain early detection of *silent hypoxia* (Teo20) (RTA$^+$21). SPIRA is a research project created during the COVID-19 pandemic to identify respiratory insufficiency via speech recordings (CGC$^+$21) with Machine Learning models. It involves doctors, speech therapists, linguistics, and computer scientists from different fields (artificial intelligence, computer music, software engineering) and was idealized as a tool for doctors and nurses to pre-diagnose this illness.

During the first stage of the project, the authors of (CGC$^+$21) created a ML model that could be used to identify respiratory insufficiency with reasonable accuracy, enabling a new phase of the project focused on the development of a more sophisticated generation of models.

This new generation now should be validated by carrying out inference experiments with patients from different hospitals under the supervision of medical personnel. These professionals are not expected to have familiarity with ML related coding tools, therefore a system is needed to properly provide inference features for them. Nevertheless, SPIRA does not have a system capable of providing this functionality yet.

The main goal of this research is to allow physicians, nurses and other related hospital personnel to perform a pre-diagnosis of respiratory insufficiency using SPIRA models. Milestones to achieve this objective are described in more detail later at chapter 4.

This monograph describes the planning, development and implementation of the inference system created to serve this new generation of SPIRA models and it is subdivided as follows: chapter 2 describes the theoretical basis, chapter 3 shows a preliminary implementation of the system, chapter 4 describes the architectural features of the inference system, **??** describes the implementation details, chapter 8 describes how the system was tested and chapter 9 shows the results obtained.

Chapter 2

Concepts

2.1 Intelligent Systems

2.1.1 Use cases

An intelligent system is a software system that uses artificial intelligence, often produced via machine learning approaches, offering leverage and support to it in order to achieve an objective (Hul18).
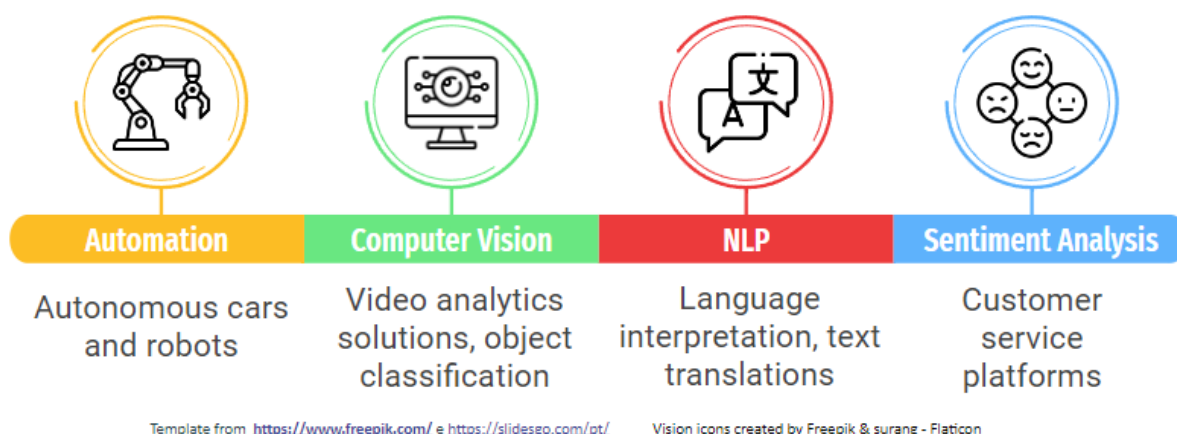
Figure 2.1: *Intelligent System applications.*

It is appropriate when either there is no analytical solution for a certain problem or the analytical alternative is too expensive given the limited computational resources available, in which cases an empiric solution via machine learning is more suitable and affordable.

Examples of such problems are:

- problems that require a considerable amount of work.

- open-ended problems, which continue to grow over time.

- time-changing problems, where the right answer changes over time.

- problems that are intrinsically hard.

Detecting respiratory failure is an intrinsically hard problem as already indicated in (TLJ20). The symptom may reflect differently depending on the disease associated with it, such as in the case of *Silent hypoxia*. This is also an indication that the diagnosis is susceptible to changes over time.

As commented in chapter 1, the use of machine learning models showed to be a good solution for pre-diagnosing respiratory insufficiency, therefore an intelligent system is now needed to provide the desired inference feature and manage the life cycle of the SPIRA models.

2.1.2 MLOps

In the field of software engineering, DevOps practices are used to shorten the development life cycle, providing continuous delivery to deploy new features more efficiently (EGHS16).

Unlike general software features, ML models are entities whose behaviors are determined by a training stage and therefore cannot be controlled, since they depend on data that is only available at prediction time. This leads to many problems such as the lack of reproducibility and difficult deployments as indicated in (Hut18; TOS+20).

For this reason, DevOps culture in the Machine Learning field has originated a new set of practices called MLOps, which aims to deploy and maintain machine learning models reliably and efficiently (TOS+20).
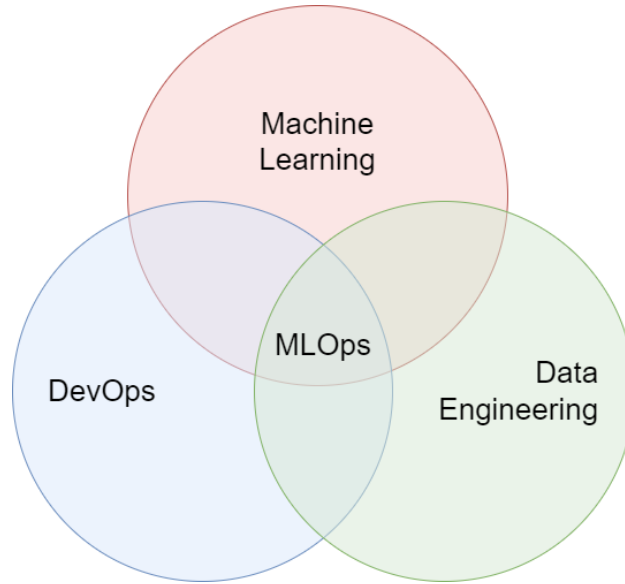
Figure 2.2: *Origins of the MLOps set of practices*

A ML model is an artifact that does not depend on a particular set of data. Therefore, it is possible to store it, keep track of its version and reproduce it when needed, which is a particularly relevant feature for the model validations that will be conducted in the SPIRA project through the inference system. It is also important to notice that the life cycle of models is independent of the software life cycle of the intelligent system, hence both can be developed concurrently.

In DevOps, CD pipelines are created to automatically deploy new software. Likewise, an ML pipeline can be created to automate the deployment of ML models by storing its artifacts in production.

The lack of a defined process are cited as one of the main challenges faced in Machine Learning (NAO+19). A ML pipeline not only shortens model creation time span considerably, but it is also a way to standardize model development procedures, which would further improve the overall productivity for the creation of new models inside the SPIRA project.

Nowadays, there are several MLOps solutions such as MLFlow, KubeFlow and Comet that can be used to implement each of the practices presented previously. Besides storing model artifacts and creating an ML pipeline, they can also be used to run experiments with the models and monitor their performance in production.

Preliminary implementations not adopting such an MLOps solution are described in chapter 3. These experiences served as the motivation to search for a more sophisticated procedure to deploy

the SPIRA models for the definitive system and its planning will be further described in chapter 4.

2.2   Microservices Architecture

A microservice is a small autonomous service that performs a single functionality. A microservices architecture is an architectural pattern where the system is composed by microservices (Bon16). Implementing a microservices architecture offers many benefits to the system:

- **Modularity:** contrary to the monolithic multi-functional paradigm, if a service contains a set of well-defined functionalities, the architecture can help modularizing them into self-contained microservices and avoid turning the system into a "big ball of mud" (FY97).

- **Horizontal Scalability:** a service may be scaled according to its load, without the need to change the remaining system (DLT+17). Hence, resources may be distributed more efficiently to optimize the performance of the application.

- **Feature Delivery:** with each microservice having its own isolate deploy, features can be implemented more frequently, because developers will not interfere with one another while working in different microservices.

- **Poliglotism:** isolation between services also allows each of them to use the technology that is the most suitable for their functionality, as long as they respect the established communication protocols.

Certain trade-offs need to be taken into account when adopting a microservices architecture. In a distributed system such as one using a microservices architecture, additional uncertainties are added due to the lack of a consistent and reliable shared memory between nodes. Hence, services are not able to fully rely on one another to make a decision.

This uncertainty is very present in the concept of time. Timestamps and clocks are unique for each service, therefore a distributed system has no universal concept of "now" that can be used to synchronize them. One solution is to accept this uncertainty and develop strategies to cope with it.

Examples of them are: logical clocks (such as vector clocks), eventual consistency (e.g. certain NoSQL databases) and making sure the communication protocols are associative (batch-insensitive), commutative (order-insensitive), and idempotent (duplication-insensitive). As it will be presented later in chapter 4, these properties are not only important for planning the architecture, but also for choosing the right tools to implement the system.

The inference system will contain multiple model microservices, each containing a SPIRA model, all of them communicating with another microservice responsible for providing an inference API that will receive requests and store results. The next section shows how the trade-offs of the microservices architecture will be handled in this research.

2.3   Reactive Systems

A system is reactive when it is built based on the *Reactive Principles* (rea14). More precisely, a reactive system relies on asynchronous message-driven communication between components in an attempt to solve some of the inherent problems of distributed systems such as the ones presented for the microservices architecture.
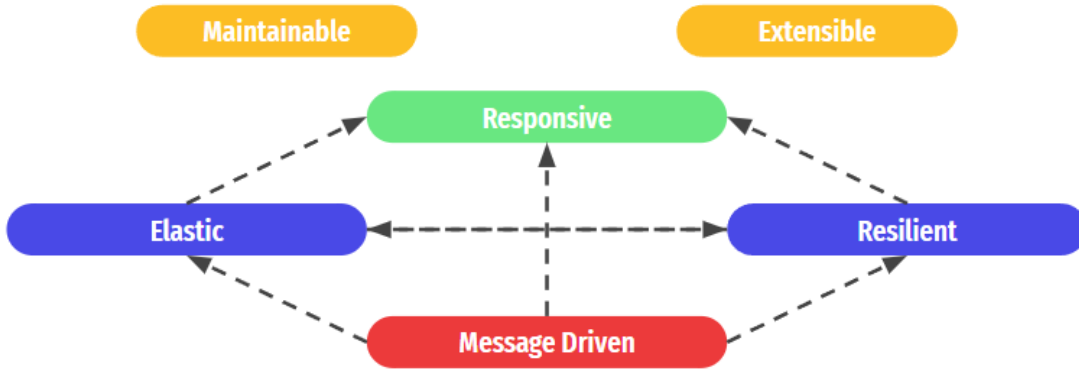
Figure 2.3: *Reactive systems benefits diagram*

The key of a reactive system is to manage the existent uncertainty between components directly in the architecture level. That is, to design autonomous components that expose their protocols and clearly define what they promise, what behavior will trigger and how the data model should be used (rea14).

Moreover, a reactive system should give each party the freedom to make its own decisions. When a node sends a call to another one, that node must have the ability to communicate back momentary degradations, such as overloads, and must have the freedom to not respond when that is appropriate. This requires the protocol between them to be asynchronous and event-driven, because component-level failures can only be contained if the communication protocol between the services anticipates the possibility of unsuccessful or late responses and neither the first nor the last interrupt their internal functioning due to these failures (rea14).

Asynchronous communication also leads to temporal decoupling, because one does not require the availability of another node to make a call. Components become even more autonomous and the system more reliable (Bon16).

Reactive systems also aim to create space decoupling by making use of network communication to avoid allocating the all its components in one specific location. This reduces the dependency on specific hardware, which are susceptible to malfunctioning and inaccessibility, and allows the nodes to be distributed in different locations to compose the system. Such freedom makes the system elastic and scalable, as it can dynamically control its resources to satisfy both momentary and permanent increases in demand for a service (Bon16).

The inference system being developed in this research will be used by multiple hospitals to validate new models created by the SPIRA project. Thus, a certain degree of volatility in the demand for each model is expected both upon their creation and also in case they outperform other models. Therefore, the intention is to follow the concepts presented above to develop a reactive system that is capable of adapting to these changes in demand.

2.4 Layered Architecture

The layered architecture is an architectural style in which a system is designed in layers, each containing components related to a specific role within the application.

While the number of layers and their roles are flexible, architectural patterns implementing a layered architecture usually contain the following three layers:

- **Presentation Layer:** responsible for abstracting inner data to the user and vice-versa. Note that this layer has no concerns regarding how the data is treated and retrieved in the application.

- **Business Layer:** responsible for executing business rules of the application. Likewise, this layer is neither involved in how the data is displayed to the user nor the origin of the data.

- **Database Layer:** solely responsible for storing application relevant data.

Besides the responsibilities of each layer, there are additional topological constraints used to maintain the service organization: a layer only interacts with two other layers and data-flow is bidirectional, i.e., either from the user to the system or the other way around.

While the layered architecture offers topological constraints to organize responsibilities in the service level, there are inherent pitfalls of the architectural style such as undesired dependencies between layers and infiltration of business logic in user interface code.

In the example layers above, the presentation layer depends on the business layer, which then depends on the database layer, violating the Dependency Inversion Principle (Mar96), as business rules depend on database technologies.

Some of the known consequences of those pitfalls are:

- developers are often not able to isolate and properly test the business logic

- the application is highly coupled with human-driven use, precluding the possibility of batch-running the system.

The violation of the DIP motivated the creation architectural patterns where the layers were organized in an onion-like shaped structure with the business logic contained in the innermost layer of the design. Examples of them are the *clean architecture* (MGB$^+$18), the onion architecture (jef08) and the hexagonal architecture (Coc).

2.5   Hexagonal Architecture

The hexagonal architecture is a layered architectural pattern invented by Alistair Cockburn in an attempt to deal with recurrent structural problems of object-oriented programming and layered architectures (Coc).

As commented in the previous section, communication between layers in a layered architectural pattern must be properly designed and managed to avoid logic infiltration. While this is crucial for the system maintainability, the layered architectural style neither specifies how this communication should be done nor how the abstractions should be converted. The hexagonal architecture solves this problem by implementing a new layer above the business layer solely responsible for managing this communication.

The architecture is then divided in three main layers: core, ports and adapters.

The core contains all the business logic of the system. It has a functional nature represented in the form of use cases, the methods of the business logic.

The adapters compose the outer layer of the architecture and they are responsible for the execution of the business logic, signaling the proper technology to take action. Examples of adapter components are database clients and API endpoints.

Finally, the layer responsible for managing the communication between the core and the adapters is the ports layer. The ports are primarily responsible for establishing the contract between the two other layers. They also convert the abstraction of an use case to the abstraction of an adapter method and vice-versa.

The ports are also divided in two categories: primary and secondary. The primary ports, also called driving ports, are the ones that drive the core calling the business logic. Example of it are REST API adapters. The secondary ports, also called driven ports, are the ones that are called by the business logic to execute some action. Examples of it are database adapters.
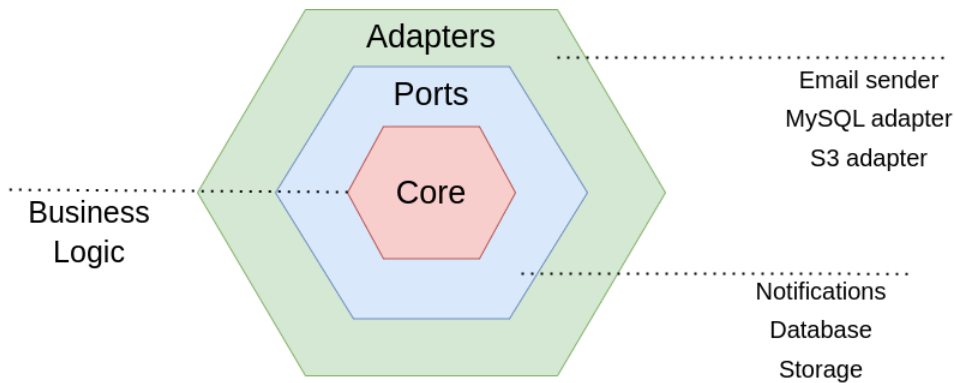
Figure 2.4: *Hexagonal Architecture diagram.* *The application contains the business logic and it communicates with the adapters through the ports. Source: https://commons.wikimedia.org/wiki/File: Hexagonal_Architecture.svg*

It is important to note two aspects of this design:

- the core is agnostic to the implementation of the adapters and ignorant of their technology.

- the ports are the boundaries of the business logic.

and they bring immediate benefits to the system. If there is a need to change the technology of a component in the system, the core code can be fully preserved and one can simply create the new adapter and connect it to the port. Additionally, the ports also become the entry point for testing the business logic, since the ports layer naturally isolates it.

The main difference from many architectural patterns to the hexagonal architecture is that most of them do not fully isolate the business logic from the adapter components. The MVC pattern, for example, implements such idea only for the primary ports with the controller layer abstraction, but not for the secondary ports.

The system being developed in this research requires many distinct technologies for its implementation, hindering the use of an architectural pattern that does not fully isolate the business logic from the technologies. The architecture also facilitates test automation with the use of mock adapters, which further helps asserting the correctness of the system as it will be discussed in chapter 8.

2.6   Progressive Web Application (PWA)

Progressive Web Applications are cross-platform compatible web applications that are able to work offline and can be added to device home screens. The term was first used in 2015 (Rus15) to describe apps that took advantage of modern browser features that enable users to convert web apps into native apps of their operating system.

Examples of such features are service workers, which are web workers used by PWAs as a proxy that checks the availability of the remote server and stores cache content for the application. This cache stores the UI content, which enables a more native-like user experience without the dependence on web connectivity. It also allows content to be loaded progressively from a initial layout provided by local storage, hence the name Progressive Web Application.

There are currently several examples of successful PWAs across the market. Examples of them are Twitter, Spotify, Uber and Starbucks Apps.

As it will be showed in chapter 2, using a PWA as the front-end tool to collect inferences has many advantages over native solutions.

Chapter 3

Preliminary Implementation

3.1 Context

The SPIRA project started with the first voice audio recording collection phase. These recordings where then used by researchers to produce their first machine learning model and prove the feasibility of symptom detection (CGC+21). Currently, a second collection phase is being conducted and the new data will be used to train a new generation of SPIRA models.

In parallel to that, there were attempts to encapsulate SPIRA models inside a service and make it available for inferences requests. While the system was successfully developed and deployed, there were many challenges faced in this phase that could either be avoided or better handled to facilitate its implementation.

The inference system being developed in this research was heavily based on the lessons learned from the mistakes made in this preliminary implementation.
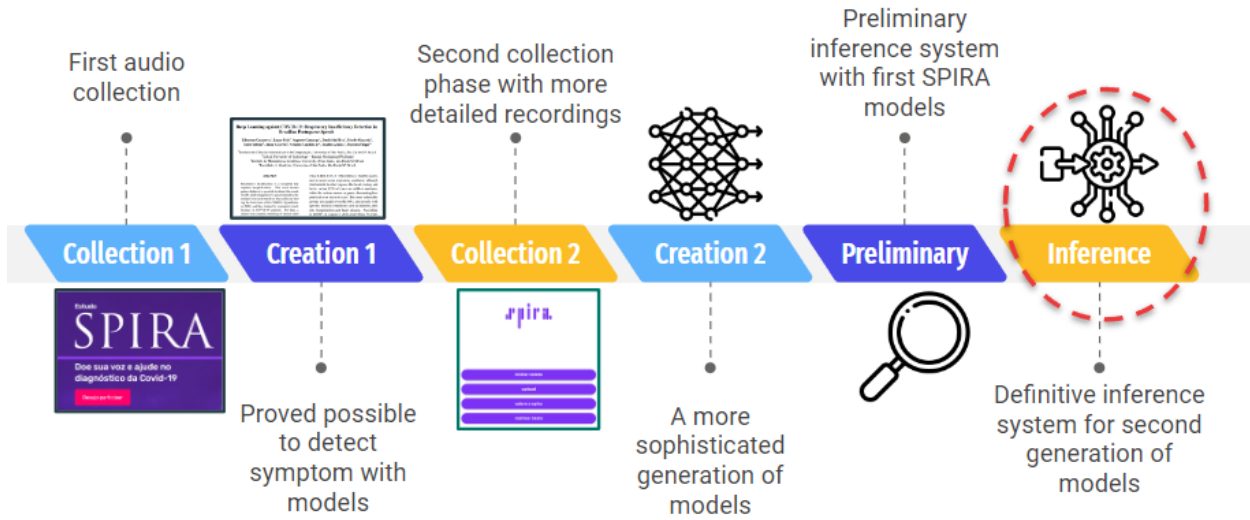
Figure 3.1: *Timeline of the SPIRA project: the present research aims to build a definitive inference system for SPIRA.*

3.2 Challenges

The model was developed in an preliminary process where the training source code was not based on any specific design pattern or convention, because rather than maintainability, feasibility was its primary goal.

The main consequence of that was the absence of a well-defined interface of the model. Hence, developers had to do create a model server adapted to create an entry point for this single model.

Particularly, input and output data processing were responsible for the great majority of the changes.

After the adjustments, developers and researchers needed to validate the model produced by the new pipeline. The goal was to ensure its predictions were coherent with the results presented in (CGC⁺21) and the validation was divided in three stages:

The first stage was to check if all the dependencies were compatible the ones used in the article. This was necessary because many machine learning libraries are under constant change due to heavy development and can change their behavior considerably between versions.

Subsequently, automated tests were created to isolate errors in the code. As stated previously, input and output processing methods were the ones which underwent through most changes, therefore special attention was given to ensure their correctness.

Finally, the third step was to do a more detailed analysis of results. Unfortunately, true reproducibility was impossible, since the researchers did not use any MLOps tool while creating their preliminary model. Therefore, there was no way to recover the exact trained model used in the experiments. Consequently, the disparity in metrics was minimized, but not fully eliminated.

After the validation process, the deployed system was able to process inference calls via HTTP protocol and retrieve prediction history. All results were stored in a local database and they were used for a more thorough analysis of the model performance afterward.

3.3   Lessons learned

The challenges faced during the preliminary implementation showed critical points of the development process and the system design that needed to be addressed for the definitive version. The sections below present the main problems and the proposed solution for them.

3.3.1   Well-defined interfaces

It was clear after the preliminary attempt that the presence of an interface between the model and the model server was crucial for the scalability of the SPIRA Inference system. This is because, without defining the boundaries of the roles of the model server and the model class, it would be necessary to adapt the model to fit the necessities of each new trained model. This would make system scaling infeasible eventually when the number of models become considerable, besides all the maintenance problems that creating the particularities of each entry point would cause.

With such an interface, the intention is to delegate the processing of input and output data for the model class, rather than the model server. Models should directly consume inference-related metadata and process it according to its own particularities before using it for prediction and the same should happen to the output value, that should also be converted back to the inference result abstraction.

3.3.2   Dependency Management

From the validation process, it was probed that dependency management needed more attention than initially planned. This step is also crucial during the containerization of the system, as dependencies are all installed in compilation time.

Another concern comes with library updates, where one might update a dependency and disregard any changes necessary to the sub dependencies, which are a potential cause of system disruption. Ideally, this process should not be done manually for the sake of long-term maintainability.

3.3.3   Automated Tests

During the preliminary attempt, tests showed to be a useful tool to detect failures in the code and they are, undoubtedly necessary for system maintenance. Despite such effectiveness, the development process of the definitive version should follow the procedures described in the TDD section to better track and test the system requirements.

Another important consideration is that the preliminary version was a mere encapsulation of the model with no reactive features in it. As a consequence, test hierarchy was not of much importance as there was no need to test the integration of the components of the system. With the new inference system architecture, nonetheless, test hierarchy does matter, as there are many levels in which the system needs to be tested to ensure its integrity, from the internal functioning of each of its components to the integration between them.

3.3.4   MLOps

Reproducibility of model behavior was infeasible during the preliminary attempt and a solution was certainly needed for model storage, versioning and monitoring. As cited in chapter 2, there are many solutions that can be used to manage the life cycle of a model and track its prediction history. This will serve as an important tool to audit the system by enabling true reproduction of a model.

Chapter 4

Architecture

4.1 Objectives

The SPIRA inference system is a reactive microservices intelligent system that pre-diagnoses respiratory insufficiency. To start the architecture plan, the milestones of the research should be unraveled first. As described previously in chapter 1, the goal of this research is to allow medical personnel to perform a pre-diagnosis of respiratory insufficiency using SPIRA models. To accomplish this objective, the research will be divided in the followinf milestones:

- **Create a solution to store new models**
  Currently, there is no default procedure to store newly trained models. The inference system needs such a solution to encapsulate SPIRA models with a server and expose them to the system.

- **Turn models accessible**
  After a model is trained and evaluated, it needs to be deployed through a service that will wrap it and make it accessible to process inference requests. Besides the technicalities of doing so, it is an even greater challenge to build a model serving infrastructure that is able to wrap any SPIRA model.

  It is important to mention that a new generation of models is under development, therefore the serving system should be scalable enough to keep up with this process.

- **Create a system to manage models and inferences**
  Once the model servers are ready, a managing system is needed to redirect inference requests, process predictions and organize the final diagnoses. The system should store all the data regarding inference metadata and results and it needs to be resilient enough to cope with the time needed to process the inferences.

- **Provide an App to be used by medical personnel to access the models**
  At last, the models are to be used by medical personnel to determine the final diagnosis, therefore a new App is needed to provide them with this access.

4.2 Context

The architecture of the entire SPIRA system is described in the image below:
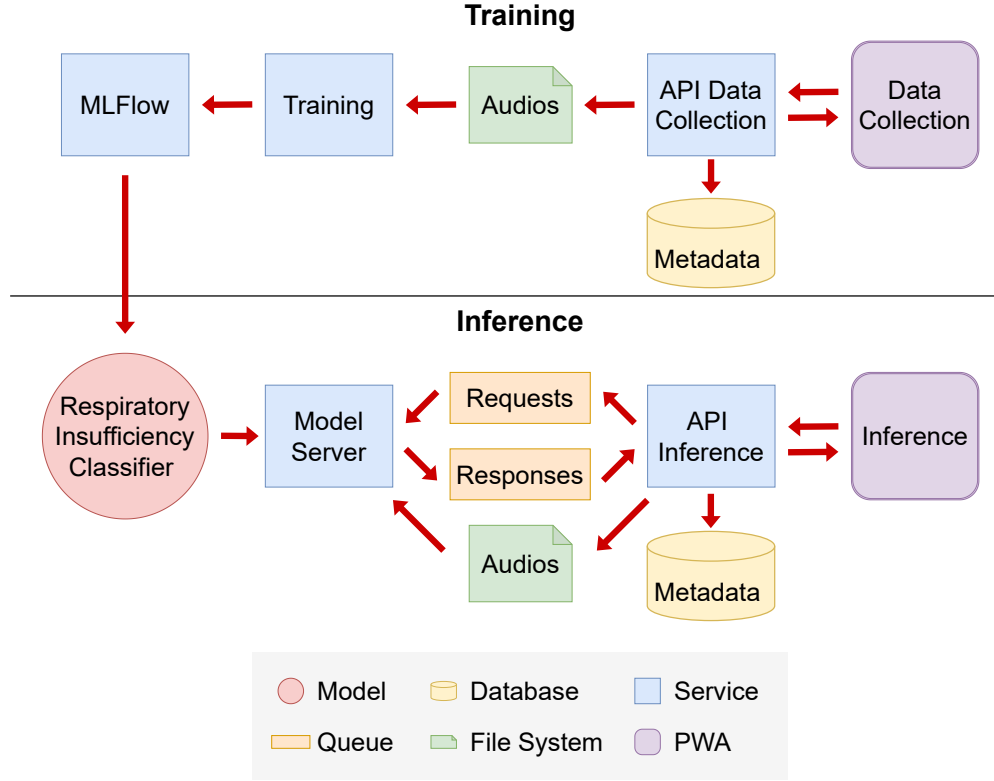
Figure 4.1: *Architecture of the SPIRA intelligent system*

The upper half of the diagram represents the training sub-system and it includes the audio data collection and storage processes, model training, model creation and finally model storage. The collection is done through a PWA specially designed to record voice audios of patients speaking specific sentences.

The data-sets will subsequently be used by the data science team to create new machine learning models. These models will be stored in a server where they will become available for use by the rest of the system.

The lower half is the inference sub-system being developed in this research and it its composed by a PWA that will send inference requests to the back-end service being developed in this project, specifying the desired model created with the data from the training sub-system. The inference request is then processed by an inference management API and subsequently sent to the respective model server carrying the target model. The final diagnosis generated by the prediction is then sent back to the API and becomes available for clients.

Since the objective of the inference system is to provide access to all SPIRA models, there will be multiple model servers deployed, each containing their own model retrieved from the model registry, enabling the API to send prediction requests to any of them.

4.3   Model Registry

4.3.1   Description

As observed during the preliminary implementation, one of the main challenges found during development was the repeatability of results, because the models used in (CGC+21) could not be replicated. Therefore, a model storage solution is crucial to follow MLOps practices during the development of the next generation of SPIRA models.

The chosen solution for the model registry is the MLFlow tracking server (mlf22). MLFlow is a platform created by Databricks to manage the lifecycle of Machine Learning models.

The platform is agnostic to ML libraries, making it compatible with any technology used to

develop Machine Learning models, including PyTorch, the most commonly used library inside the SPIRA project, and other libraries such as Keras and TensorFlow.

Therefore, if developers decide to change the technology used to create models, MLFlow will be equally compatible with the new libraries of choice.

The model registry is composed by three distinct containers:

- **MLFlow server:** The server hosting the MLFlow platform.

- **Database:** The database is used by the server to store all metadata related to experiments and model versioning.

- **Storage Unit:** It is used by the server as an artifact store for model files. The storage unit will be explained in more detail in 4.5.

Model servers will connect to the model registry during initialization time to retrieve one of the models stored in it. Details on how this retrieval is done will be explained in 4.6.2.

While the goal of this research is to use this component as a model storage solution, the registry is actually agnostic to the existence of the inference system. Developers may also use it to retrieve models for any other purpose besides inferences such as for model performance comparisons or for running independent experiments, which are also supported by MLFlow. As the server can keep track of a production version of each model, it is possible to change the tag of a model to production whenever it is ready to be deployed to the system.

4.3.2   Implementation

The registry is a docker container inspired from (reg22) and is actually a python image which downloads the MLFlow CLI and initializes a server using it. In the development repository, where the image is defined, the Dockerfile is described below:

```
1   FROM python:3.8 as mlflow_base
2
3   ENV POETRY_VERSION=1.1.13
4
5   WORKDIR /project
6
7   RUN apt-get update
8   RUN pip install --upgrade pip
9   RUN pip install "poetry==$POETRY_VERSION"
10  RUN poetry config virtualenvs.create false
11
12  COPY pyproject.toml poetry.lock .tool-versions ./
13
14  RUN poetry install --no-interaction --no-ansi --no-root
15
16  EXPOSE 5000
```

The file first pulls a python image and installs poetry package. Poetry then uses the poetry.lock file, where dependencies for the registry are declared, and uses it to install them. Finally, the internal port 5000 of the container is exposed, which is the default port used by MLFlow server.

The command to initialize the server itself is specified directly in the docker-compose file and is as follows:

```
1   'mlflow server \
2     --backend-store-uri \
```

```
3    mysql+pymysql://${MYSQL_USER}:${MYSQL_PASSWORD}@db:3306/${MYSQL_DATABASE}\
4    --default-artifact-root s3://mlflow/ --host 0.0.0.0'
```

It is necessary to specify both the metadata and the artifact storages. The metadata storage used is a MySQL DB container, whereas the artifact storage is a MinIO bucket. While the bucket is of exclusive use for MLFlow, the MinIO server used by the server is the same as the file storage solution for audio files.

The image can then be uploaded to Docker Hub and downloaded back in production to initialize the server. This process was automated with a CD Pipeline using Git Hub Actions. Details on how the pipeline was setup will be showed in section 6.3.

Models stored in the server can be revisited using the MLFlow server UI Figure 4.4.
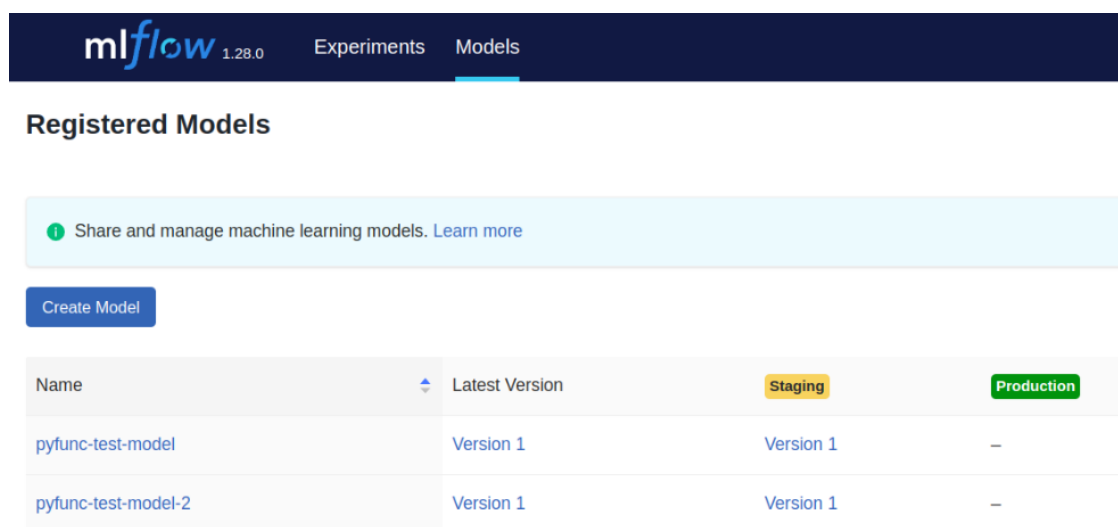
Figure 4.2: **_MLFlow UI._** _The UI can be used to track the list of models in production, the staging and production versions, and also past experiments._

The UI is exposed in the same port as the MLFlow server, hence in a private port. Developers with access to the production machine might then use it to control the staging and production versions of each model and also create new models directly from it, despite this not being the recommended method to register new models. The server supports programmatic registration of models and the ML pipeline setup to do that will be described in section 6.4.

4.4   Message Service

4.4.1   Description

As explained in chapter 2, there are numerous benefits in using event-driven communication. Mainly, the system becomes reactive, therefore more resilient and horizontally scalable. The two mostly utilized solutions to implement such asynchronous communication are message and queue services.

While the current plan is to maintain one communication stream for each model deployed in production, giving margin to implement a queue service, there are three reasons for not using queues for this system:

- **Harder model server setup:** Each newly created model will require it to be registered and a new server to be lifted with it. Using a queue service would also require a new queue to be deployed with each server, which would make this setup more complicated. Using a

message service brings the benefit of using a single broker to manage all communication stream, therefore there is no need for creating a new structure for new models introduced in the system.

- **1:N requests:** Although the system will be implemented to send inferences to single models, it might be a future feature request to send inference requests for a set of models for result comparison with the same input. A message service provides a simplified 1:N communication, as many listeners can subscribe to the same stream, while queue services would require the API to send more messages proportionally to the number of requested models.

- **N:1 updates:** Message services also enable all model servers to send their results to a single topic. If a queue service were to be used instead, each deployed model would also require an additional queue solely for sending their own updates and along with it, the number of services actively listening to incoming results on the other side of the each queue, which would again hinder the new models setup.

Once an inference request gets to the API, the request metadata will be stored in the database and it will be encoded in the form of a message. The API will be responsible for sending this message to a broker in a specific channel and will delegate the inference for the components listening to it. Once the prediction is complete, the results may be sent encoded as another message. The API will then recover this result and update the database, making it available for retrieval in the UI.

The message service to be used is NATS (nat22), an open-source solution that is easily scalable in the form of server clusters. This is a relevant factor to be considered, since the number of model servers is set to increase.

NATS maintains communication streams through a topics system. A server may publish messages in a topic to a broker and other parts of the system willing to listen to the given topic may subscribe to it using a client.

In the inference system, there will be a single broker with each model subscribed to an exclusive topic. Once the API receives a request, it will retrieve the correct channel to be used in the database and will then send the message in the given topic. The model server subscribed to it will then proceed with the prediction.

Updates will all be sent to an updates topic. The API will subscribe to it while all model servers will send messages through it.

NATS event-driven communication will require two distinct adapters in both the API and the model server: A message service adapter and a listener adapter.

The message service adapter will be the point of communication between the servers and the broker. It will be responsible for encoding and sending inference requests to a certain topic. This section of the adapter will be called by the use cases triggered by the web-server. The incoming inference requests will schedule the messages as shown in the diagram below and a successful status will be returned to the front-end in case the message is successfully sent.

The same adapter is also responsible for receiving and decoding messages of a given topic. Receiving a message on push mode, i.e. actively wait for messages in a given topic, depends on having a process exclusively for calling the receiver method. This process is created by the listener adapter.

The listener will call the business logic use case of receiving a message in a given topic, the message service adapter will then be called to wait for a message and, once arrived, the received message will trigger the database update. Once completed, the method ends and the listener restarts this process.
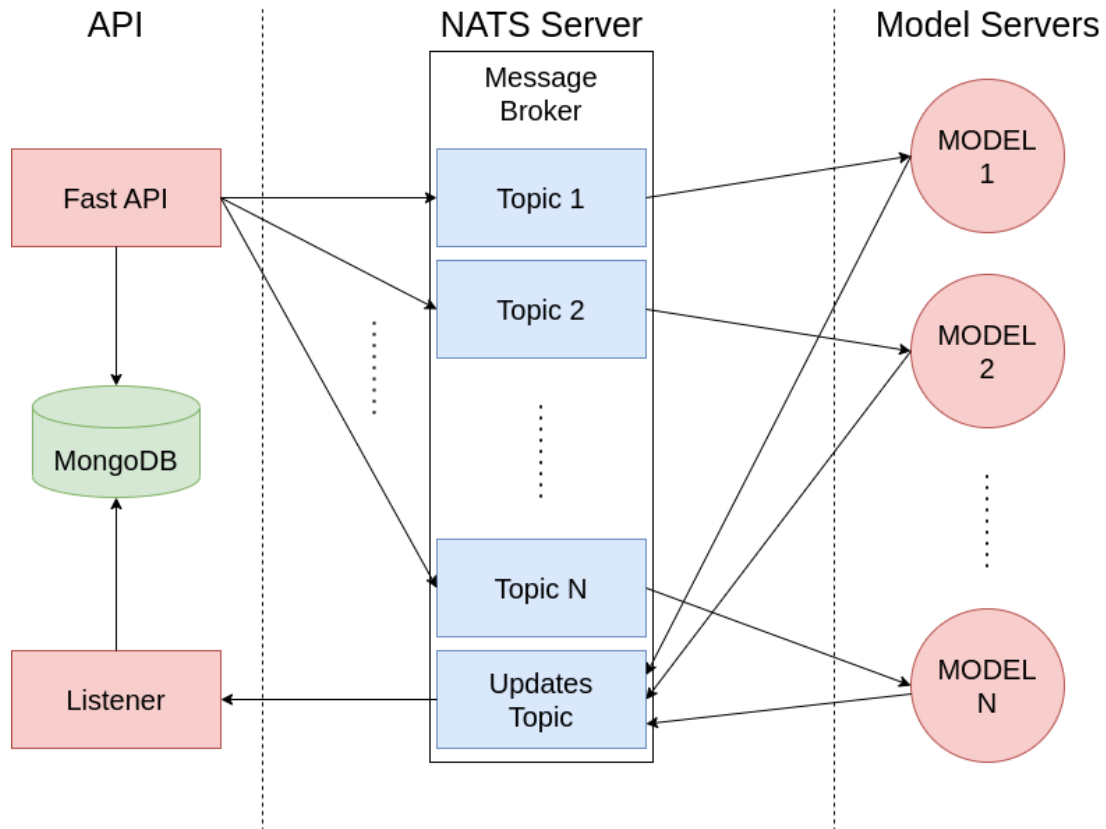
Figure 4.3: *NATS topic diagram. The API sends inference messages to each topic*

4.4.2   Implementation

The NATS server is a docker container pulling the NATS image. The service in the docker-compose file is described below:

```
1  nats:
2      image: nats
3      restart: always
4      ports:
5        - '${NATS_PORT_1}:${NATS_INNER_PORT_1}'
6        - '${NATS_PORT_2}:${NATS_INNER_PORT_2}'
```

NATS requires two ports to be specified. By default, 4222 is the one used for clients subscribing and publishing messages to the broker and 8222 is an information port UI where connections and messages can be monitored. The ports are then mapped to private ports in the local machine, where they become available for users with access to production.

Figure 4.4: *NATS UI.* *The UI can be used to perform server health-checks, or check live connections and subscriptions.*

NATS also exposes a third port 6222 for clustering support, however this feature will not be implemented in this research an may be considered as a future improvement for the messaging system.

4.5 Storage

4.5.1 Description

The file storage has two distinct roles in the system:

- **Audio recordings:** Inference requests contain audio recording files that are crucial for result prediction. As explained, the model server and the API server are separated by a message broker. NATS, as many other message services, have a message size limit of 1MB, which is not enough for the four audio recordings. Thus, a file storage solution where the API is able to share recording with the model servers is necessary.

- **MLFlow Artifacts:** MLFlow server delegates data storage functionalities to other components. One of the delegations is for artifact storage, which require a file storage service.

MLFlow documentation recommends an S3 bucket or similar file system as the default artifact storage, therefore the solution should be preferably compatible with the S3 service. It is also of the interest of the SPIRA project to maintain all audio recordings stored locally, as these recordings may also be used for training and auditing in the future.

Due to these requirements, the chosen solution was MinIO simple storage (min22). The MinIO solution is compatible with the Amazon S3 cloud service and stores all its contents in a local volume.

Like the S3 simple storage, MinIO organizes the stored files as objects inside buckets. The system will contain two buckets. The first bucket will be used by the API to store the incoming audio files in directories referenced by the inference ID. Model servers can then access these files using MinIO client. The second bucket will be used as the default artifact storage of the model registry and will not be accessed directly by any part of the system other than the registry itself.

Similarly to the message service, a simple storage adapter will be used by the API and the model server to interact with MinIO server. The API will require a method to store and delete files and the model servers will only require a method to retrieve them.

Storing should happen before the inference message is sent to NATS server, while deletion will only happen when the inference update arrives through the listener. This way, it is guaranteed that the model server will find the necessary files in MinIO by the time it receives the request and starts the prediction.

4.5.2   Implementation

The file storage server is a docker container pulling the MinIO image. The service in the docker-compose file is described below:

```
1  minio:
2      image: minio/minio
3      restart: always
4      ports:
5        - '${MINIO_PORT_1}:${MINIO_INNER_PORT_1}'
6        - '${MINIO_PORT_2}:${MINIO_INNER_PORT_2}'
7      command:
8        - server /data
9          --console-address ':${MINIO_INNER_PORT_2}'
10         --address ':${MINIO_INNER_PORT_1}'
11     volumes:
12       - minio_data:/data
13     env_file:
14       - .env
```

MinIO Server requires two distinct ports. The default container ports used by MinIO are 9000 and 9001 and they can then be mapped to ports in the local machine. The first port exposes a file management API that is used by other services to manipulate the files, while the second port exposes a UI that can be used to revisit stored objects in the server, as shown in Figure 4.5. Both ports are private, which means that only users with access to the production server have access to it.
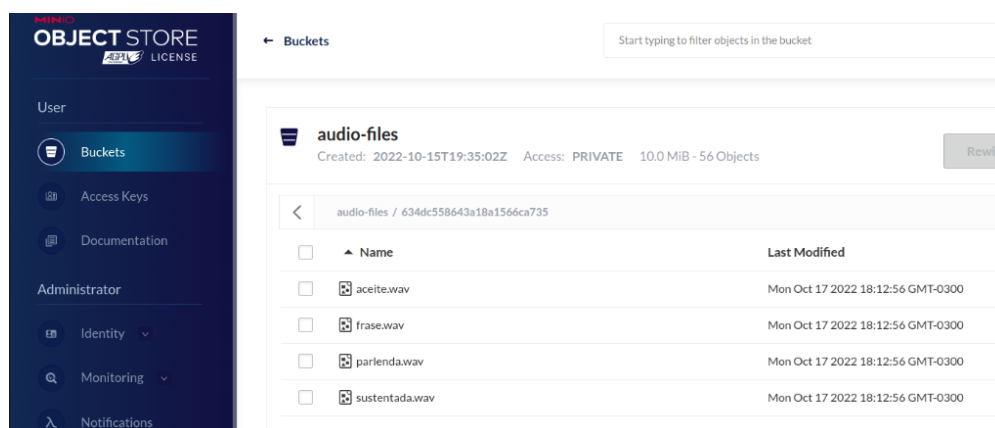
Figure 4.5: *MinIO Server UI. The audio files are stored in a bucket and can be consulted in the directory named with the corresponding inference id*

MinIO stores all its contents in /data by default, which is a docker volume minio_data in the local machine, hence the data is preserved even if both the image and the container are both deleted.

4.6   Back-end Services

The implemented services, i.e. the model server and the API, were implemented in Python. There were trade-offs taken into account to choose the language.

The development of an intelligent system requires a language with great support for Machine Learning, as it will be constantly dealing with model inferences. Python is currently one of the most popular language for ML, making it a great choice for the project due to its large community support and wide variety of frameworks and tools related to the area.

Particularly, the SPIRA models are developed with PyTorch library, which is exclusive for Python language, therefore the use of Python is unavoidable to some extent, since the interface between the model and the inference system needs to be defined.

In addition, building a project in the same language not only guarantees that there will not be compatibility issues when interacting with the model, but also provides a variety of functionalities from the dependencies that would otherwise not be available using another language.

While Python is not the most suitable language for Object Oriented Programming, which is necessary to implement the hexagonal architecture, there are tools that may be used to facilitate its implementation, such as Pydantic, used to control variable typing inside the core of the services.

Thus, the ML compatibility benefits of using Python far out-weight the cons, as they can be easily handled to not jeopardize the implementation of the services.

The final tools used in the implementation are described in Figure 4.6.
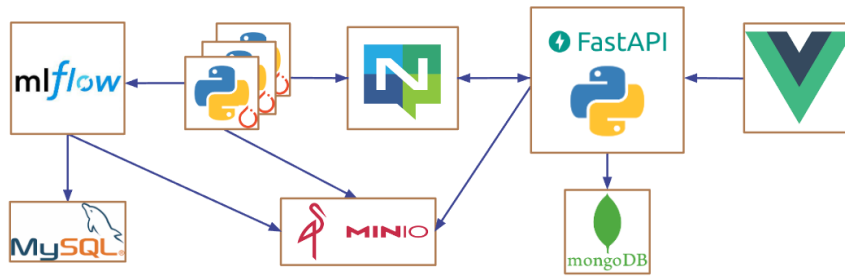
Figure 4.6: *Diagram of the tools used to implement the inference system.*

4.6.1   API

The inference management API service is the central component of the inference system. It is responsible for receiving incoming inference requests from the PWA. It will serve as a gateway, sorting the requests and sending each of them to the desired model server through the message service. The system may then be used to compare the performance of each model, validate them and filter the best generations for further study.

The service is composed by two distinct threads. The first thread is a web server which will be publicly exposed, from where the UI client will communicate with the back-end system. The second thread is a listener loop which is subscribed to a NATS topic. This topic is where all the incoming results from inference models will be published. The listener thread will receive the messages and update the database with the corresponding predictions for each inference.

The API also contains a database to control user, model and inference related data. The database solution to be used for the project is mongoDB. The reason for this choice is the fact that mongoDB is one of the most widely used NoSQL database solutions.

As explained in chapter 2, NoSQL databases use eventual consistency to prioritize availability rather than consistency, which is a strategy to deal with time uncertainties in distributed systems. While this means that prediction updates may not propagate instantaneously, the database is guaranteed to eventually update the inference with the final diagnose and the diagnose will be available in the UI.

Further details about the implementation of the API will be given in chapter 5.

4.6.2   Model Server

The component responsible for retrieving a model from the MLFlow server and making inferences with it is the model server. As explained in section 4.2, the system will contain multiple model servers, each wrapping a model from the registry.

The model server first retrieves the model from the registry and subscribes to a specific NATS topic in initialization time. The API publishes messages in this given topic whenever it processes inference requests for that particular model. The model server, which continuously listens for messages incoming from the NATS server, receives the published message, processes the inference using the retrieved model instance and then publishes another message containing the diagnose in the updates topic in which the API subscribed.

As explained in chapter 2, time uncertainties need to be properly handled in order to guarantee that inferences are being correctly processed by the system. One of the ways to cope with time uncertainties is to ensure that the operations are associative (batch-insensitive), commutative (order-insensitive), and idempotent (duplication-insensitive). In the current research, operations are inferences made by model servers.

Note that if there are multiple model servers loaded with the same model and therefore, listening to the same message topic, each model server instance will process the prediction and send the corresponding result for an inference, referenced by its id. Since both the model and the input is equal in all instances, any of the obtained results is a valid one and therefore the last of the updates in the inference result, which is the one who will persist in the database, is valid. Hence, the operation is batch-insensitive.

The model ID is part of the primary key of the inference entity, which means that an inference relates to exactly one model of the system, therefore no particular order of inference requests will give different final persisted data in the database. Hence, the operation is commutative.

Finally, if the same inference is processed multiple times, e.g., an inference request somehow triggers multiple messages for the same model server, the effect is similar to having two distinct instances of the same model, in which case the final result is still valid. Therefore the operation is also idempotent.

With those three properties, we may ensure that, if the model prediction is correct, the system will also correctly inform the result to the user.

Further details about the implementation of the model server will be given in chapter 6.

4.7   UI

The UI Application is the PWA to be used by medical personnel. Users, as explained, will send inference requests through the UI and will also visualize inference results through it. The PWA will communicate with the collection server to retrieve form related data, maintaining the same flow as the collection App, and will also retrieve model and inference related data in the new inference API server.

Currently, there exists a collection PWA which is being used for the new collection phase of the SPIRA project. The planning of the application passed through UX reviews and is already being used by medical personnel. Since the collection and inference processes are identical to each other. Using the same UI structure for the inference app not only helps with the adaptation of users to the new app, but also guarantees that no bias is added to the inference process due to changes in the UI layout.

Hence, the currently existing data flow used by the collection PWA may be used to start sending inference requests to the inference API instead of sending collecting data to the training back-end.
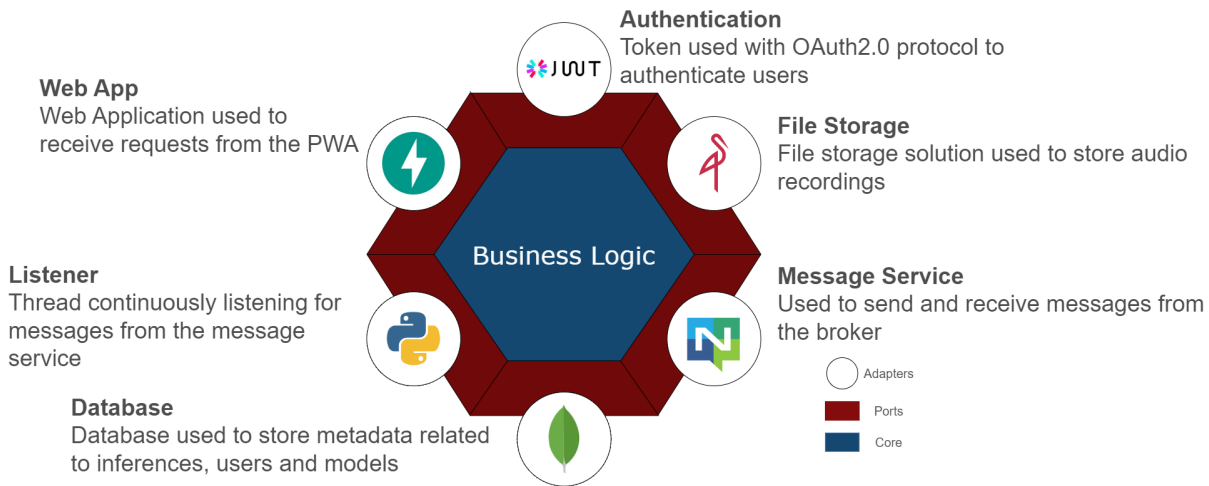
Furthermore, also due to the similarities, the collection PWA may be adapted to be deployed in either collection or inference mode. This way, the same project can be re-utilized for the development of the inference PWA.

Chapter 5

API

The architectural pattern used in the API is the Hexagonal Architecture Figure 5.1. The adapters implemented in it and the core business logic are explained in more detail in the following sections.

Authentication
Token used with OAuth2.0 protocol to authenticate users

Web App
Web Application used to receive requests from the PWA

File Storage
File storage solution used to store audio recordings

Business Logic

Listener
Thread continuously listening for messages from the message service

Message Service
Used to send and receive messages from the broker

Database
Database used to store metadata related to inferences, users and models

Adapters
Ports
Core

Figure 5.1: *Architecture of the API service.*

5.1 Core

The core of the API is where the business logic is contained and it is composed of two components: the entities and the services.

The entities are the definitions of the classes used inside the business logic. They were defined using the library Pydantic, a data validation Python library that enforces the use of type-hints. The package will be used by the use cases to ensure that types are being followed correctly during run time.

The services, or use cases, should be immutable to the environment conditions (Coc), therefore the core does not use environment variables. They are rather all utilized inside the adapters. Hence, use cases are reliant solely on the inputs provided by the primary ports to understand the circumstances of the call. Through this reasoning, the functional paradigm becomes more suitable to them in detriment of object-oriented programming. Thus, the core should be fully functional.

With such a core, environment changes do not interfere will its business rules, which facilitates the deployment of the system and promotes space decoupling, as desired due to reactivity. Note that a functional core also ensures that a change in one of its use cases will not be disruptive to other sections of the business logic, which is beneficial for future feature requests directly related with the core.

The following example is the use case called by the endpoint showed previously for user creation:

```
1   def create_new_user(
2       authentication_port: AuthenticationPort,
3       database_port: DatabasePort,
4       user_form: UserCreationForm,
5       token: Token,
6   ) -> None:
7       """
8         Docstring.
9       """
10      try:
11          if not authentication_port.validate_token(token):
12              raise DefaultExceptions.credentials_exception
13
14          _validate_new_user(database_port, user_form)
15
16          new_user = UserCreation(
17              username=user_form.username,
18              email=user_form.email,
19              password=user_form.password,
20          )
21          new_user.password = authentication_port.get_password_hash(new_user.password)
22          database_port.insert_user(new_user)
23
24      except LogicException:
25          raise
26      except:
27          raise LogicException(
28              "cound not create new user", status.HTTP_500_INTERNAL_SERVER_ERROR
29          )
```

The method first authenticates the call and proceeds with the user form validation. Once the form is validated, the new user is then inserted in the database. If failure for any reason happens during this process, the exception is caught and informed to the requestor through the status code. Likewise, each endpoint in the router calls their respective service.

5.2   Web Server

The framework used to implement the web server was FastAPI due to its better performance in comparison with Flask. FastAPI also has built-in support for some of the adopted solutions, such as the OAuth2.0 protocol that is used in the authentication system.

Additionally, FastAPI is able to automatically convert request body contents as Pydantic Base-Models. This feature is particularly useful for POST requests such as user and inference creations.

Additionally, users are only allowed to perform inference operations within their own scope, that is, /users/:user_id. Any request from a third-party user to an endpoint out of this scope will generate a Forbidden Operation response.

The endpoints are abstracted in the Hexagonal Architecture as an adapter. Each endpoint is declared as a python function in FastAPI. The router then calls the core method that will execute the corresponding service. Following the same example of user creation, the implementation of the /users endpoint is described below:

```
1   @router.post("/")
2       def create_user(
```

```
 3          user_form: UserCreationForm,
 4          token_content: str = Depends(oauth2_scheme)
 5      ):
 6          try:
 7              create_new_user(
 8                  authentication_port,
 9                  database_port,
10                  user_form,
11                  Token(content=token_content),
12              )
13          except LogicException as e:
14              raise HTTPException(e.error_status, e.message)
15          return {"message": "user registered!"}
```

A POST request in /users will call the method above, which will then call the create_new_
user method inside the core. The core method will add the entity in the database through the
database port and then the mongoDB adapter and will acknowledge the web server regarding the
success of the operation. The sequence of calls is described in more detail in the sequence diagram
Figure 5.2.
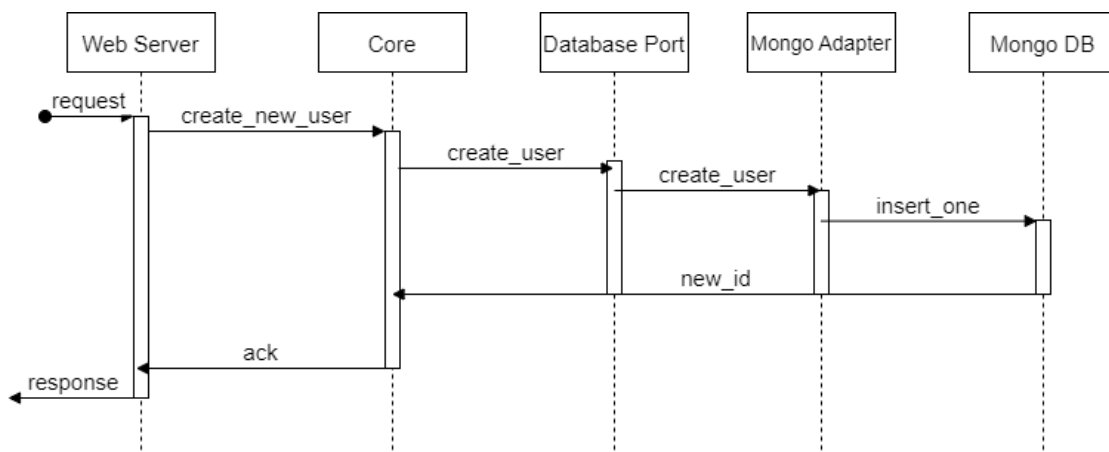
Figure 5.2: *User creation sequence diagram.*

If the request is processed successfully, a response is returned with 200 as the default status
code, otherwise, a LogicException object is thrown by the Core and the router converts it to an
HTTP response.

The implemented server supports requests in the following endpoints:

Request	Endpoint	Description
GET	/users/:user_id	Retrieves the user entity from DB by the user ID
POST	/users	Creates a new user
POST	/users/auth	Authenticates user credentials
POST	/models	Creates a new model
GET	/models/:model_id	Retrieves the model entity from DB by the model ID
GET	/models	Retrieves a list with all models
GET	/users/:user_id/inferences/ :inference_id	Retrieves the inference entity from DB by the inference ID
POST	/users/:user_id/inferences	Creates a new inference
GET	/users/:user_id/inferences	Retrieves the list of inference entities of a user
GET	/users/:user_id/inferences/ :inference_id/result	Retrieves the result of an inference

5.3   Authentication

In contrast with the training system, inference related operations using SPIRA models are not designed to be publicly available. The intended use of the system is restricted to medical personnel and therefore an authentication mechanism is necessary in order to satisfy this requirement.

The protocol to be used is OAuth 2.0. This protocol requires users to send their credentials to an especial endpoint (in the implemented web server, this endpoint is users/auth), where they will be validated and responded with a token. The users may then use this token to access the inference system and make requests using it.

The format of the token may vary according to the projects decisions. The token pattern to be used in this research is the JSON Web Token, mostly known as JWT (jwt22). JWT is composed of three distinct strings. The first part of it is the header, where details regarding the algorithm used to encode the token are stored. The second part is the payload, that may include any data relevant for the application. The last section of the token is the signature and it is the most relevant part for authentication. The server uses a private key along with the header and the payload to encode the signature when creating the token for a user. In case a malicious request is sent with a fake token, the server is able to validate by checking if this signature is correct given its private key, which is supposedly not accessible to the requestor.

In accordance with the architecture pattern adopted, the authentication mechanism is implemented as an adapter. The business logic is then responsible for implementing the OAuth2.0 protocol, while being completely agnostic to the JWT implementation. The adapter on the other hand, is solely responsible for correctly implementing JWT authentication, while also being ignorant of the fact that the OAuth2.0 protocol is being used. As a result, the protocol and the token pattern may be changed for newer and more sophisticated versions in the future without interfering with each other.

The core method to generate the token upon an authentication request is as follows:

```python
def authenticate_and_generate_token(
    authentication_port: AuthenticationPort,
    database_port: DatabasePort,
    username: str,
    password: str,
) -> Tuple[str, Token]:
    try:
        user: User = _authenticate_user(
            authentication_port, database_port, username, password
        )
```

```
11          if user is None:
12              raise
13
14          token = authentication_port.generate_token(
15              data=TokenData(username=user.username)
16          )
17      except:
18          raise DefaultExceptions.user_form_exception
19
20      return user.id, token
```

Users credentials are first authenticated and, in case they are valid, a token is generated through the authentication port and adapter. The method proceeds returning the token. Note that in any moment the core is informed that the token is a JWT.
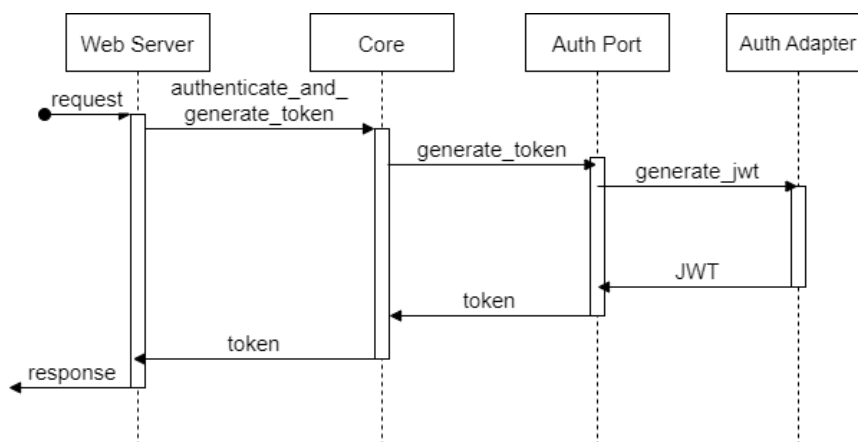

Figure 5.3: *User authentication sequence diagram.*

The sequence of calls is detailed in Figure 5.3.

5.4   Database

5.4.1   Description

The database stores information regarding four main entities: users, models, inferences and results.

The users collection stores the hashed passwords of the accounts. The hashed passwords are then compared with login credentials to authenticate the access and generate the JWT token that will be used in subsequent requests of that account.

The models collection is related to the models that are available for use in the system. It also contains the NATS topic in which the API will send inference request messages for that particular model. Model servers retrieving this a particular model should then subscribe to this topic in order to receive the requests properly.

The inferences collection contains all metadata related to inferences. The entity is currently defined as a Pydantic BaseModel inside the Core as:

```
1   class Inference(BaseModel):
2       id: str
3       status: str
4       user_id: str
5       created_in: str
```

```
 6       model_id: str
 7       rgh: str
 8       mask_type: str
 9       gender: str
10       covid_status: str
11       local: str
12       age: Optional[int]
13       cid: Optional[str]
14       bpm: Optional[str]
15       respiratory_frequency: Optional[str]
16       respiratory_insufficiency_status: Optional[str]
17       location: Optional[str]
18       last_positive_diagnose_date: Optional[str]
19       hospitalized: Optional[str]
20       hospitalization_start: Optional[str]
21       hospitalization_end: Optional[str]
22       spo2: Optional[str]
```

While the definition of inference is already set for the current phase of the SPIRA project, this definition is still subject to change. The current definition of an inference is actually an improved version of the preliminary implementation, which implies that future versions may also include additional data.

This volatility in the definition is also one of the reasons why a NoSQL database is more suitable for the purposes of this research. A relational database would require constant back-fills of missing information each time the definition of an inference is changed, while a NoSQL database, such as MongoDB, is able to support all the definitions at once.

The results collection contains the results of an inference and its entities contain the output of the model along with the final pre-diagnosis given by it. They are stored in a different table, because despite the current implementation mapping an inference request to only one model, that is, one result, the system may be adapted in the future to trigger multiple model predictions with one inference request. In this case, it is useful to maintain the separation between the two entities so that this feature can may be implemented more easily.

5.4.2   Implementation

The database service is a docker container pulling the mongo image. The image does not provide means to visualize the data stored in the DB such as with NATS and MinIO, however there are separate solutions that can be used to provide the direct access to mongoDB. The solution used in this case was a new container pulling the mongo-express image, which was then exposed to a private port of the local machine.

```
 1   mongo:
 2       image: mongo
 3       restart: always
 4       env_file:
 5         - .env
 6
 7   mongo-express:
 8       image: mongo-express
 9       restart: always
10       depends_on:
11         - mongo
12       env_file:
```

```
13        - .env
14      ports:
15        - '${ME_PORT}:${ME_INNER_PORT}'
```

The UI not only provides a visualization of the data, but also a way to manually add and change entities in the collections of the database.
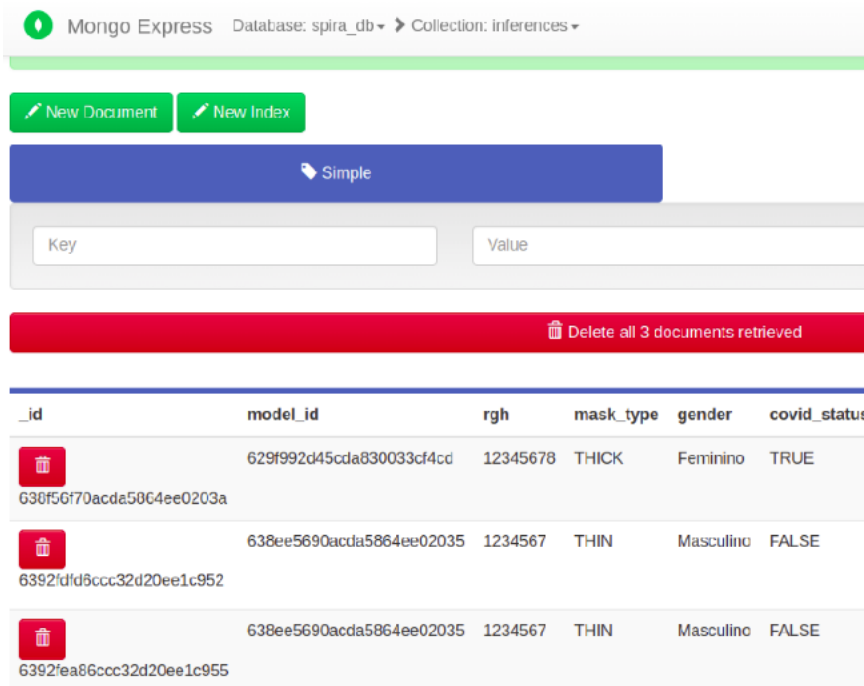

Figure 5.4: *Mongo Express UI. The UI can be used to directly interact with the database.*

5.5 Inference Requests

The main data flow of the API are the inference requests. Upon receiving a POST request at `users/:user_id/inferences`, the API will receive the request and store a new inference entity in the database, following the sequence diagram described in Figure 5.5.
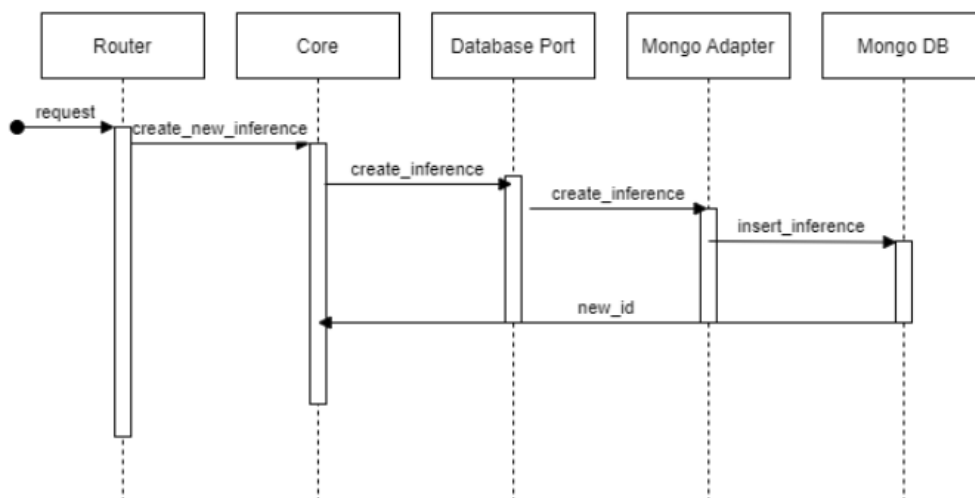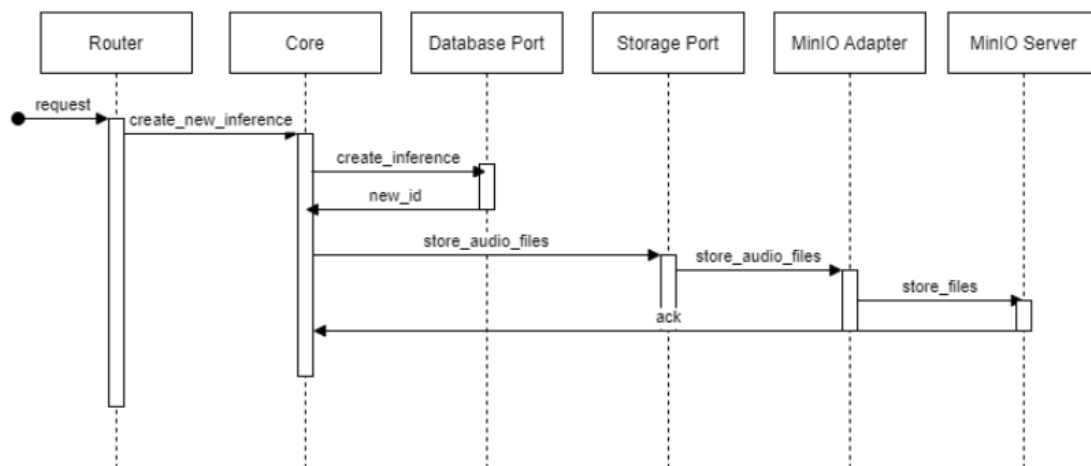

Figure 5.5: *Database sequence diagram for inference requests. Once inference requests are received, the web server calls the core, which then calls the database port to store the entity in the Mongo DB.*

Once the entity is inserted in the database, the new inference ID is returned to the core and the method will proceed storing the audio recording files from the body of the request in the MinIO server, following the sequence described in Figure 5.6.

The inference request is currently composed of four distinct audio recordings:

- **Aceite:** Verbal acceptance of the terms of service

- **Sustentada:** Sustained vowel for as long as the subject can hold on

- **Parlenda:** A nursery rhyme presented by the app

- **Frase:** A sentence presented by the app

The four audio files are stored in the *audio-files* bucket, at a directory named with the ID obtained from the database insertion.

Figure 5.6: *File storage sequence diagram for inference requests. Audio files are stored in MinIO server to be retrieved by model servers later.*

Finally, once the storage port acknowledges the core that the files were inserted, the method proceeds publishing a new message in the topic of the requested model. The sequence is described in Figure 5.7. The message body contains the encoded inference entity described in section 5.4.
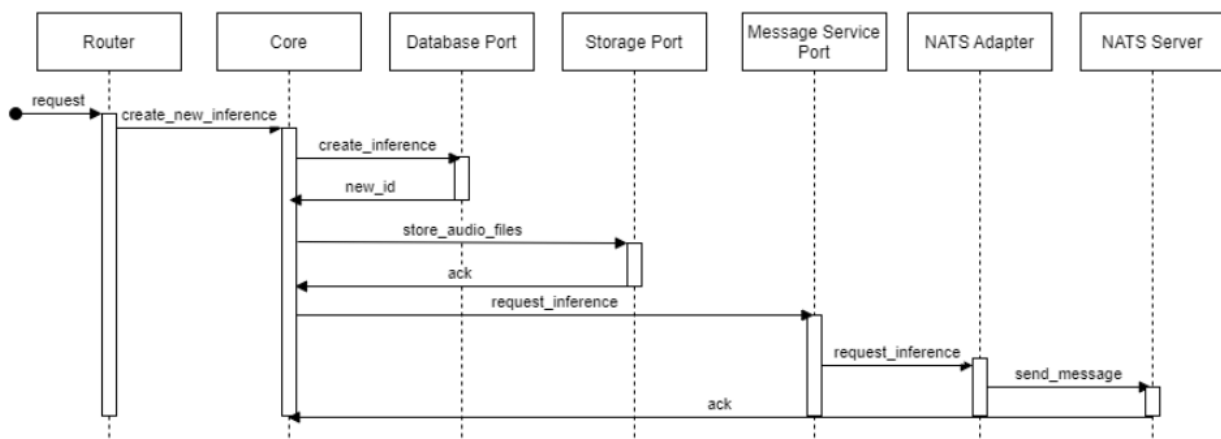
Figure 5.7: *Message service sequence diagram for inference requests. Published message will be received by a model server subscribed to the model topic.*

As explained in section 2.3, a microservice has no information regarding the health of another microservice. While the model server is able to decide whether to process a request or not due to momentary degradations, the API is prepared to receive a late or unsuccessful response from the model server. For this reason, the API sends a successful response to the client once the message is successfully sent to the NATS broker, without waiting for an acknowledgment that the inference was received or processed. The API then *reacts* to incoming responses when a model server does manage to process an inference request, but does not stop its execution in case no response is obtained.

5.6   Deployment

The docker image of the API is built through a Dockerfile that pulls a python image. The resulting image can then be uploaded to Docker Hub (doc22), from where it can be pulled to assemble the production system.

The production repository is located at (inf22). It contains two docker-compose files, one solely for the model-server service and another containing all the rest of the back-end system. This separation is necessary so that model servers can be added and removed from the inference system without the need to restart the entire system.

The deploy is hosted at the *Inspirada* server at https://inspirada.ime.usp.br. The production repository was cloned in the machine and a .service file was created to run the API service as a systemd unit.

A continuous delivery pipeline was set up with Git Hub Actions to automatically upload images to Docker Hub upon a tag creation in the development repository.

Git Hub Actions are configured through .yaml files in the `.github/workflows` directory inside the repository. The tags of the images were configured to be the same as the pushed tags in the repository.
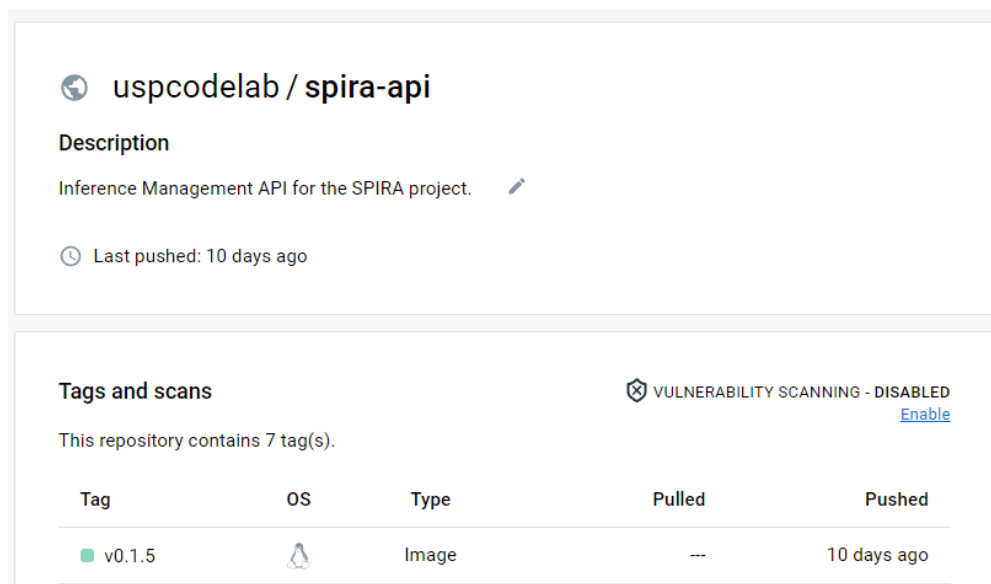
Figure 5.8: ***Docker Hub image repositories for the API service.*** *Images are automatically pushed to Docker Hub upon new tag creations in GitHub.*

In order to prevent man-in-the-middle attacks between the UI client and the API, an HTTPS connection was setup between the two. The TLS certification was setup using Caddy Server as a reverse proxy to the inference management API.

The main advantage of using Caddy instead of other services such as Nginx is that Caddy has TLS automation features which are considered safe and modern by the literature (ABC$^+$19), thus facilitating the configuration of a reverse proxy.

Chapter 6

Model Server

The architecture pattern used for the model server is the Hexagonal Architecture and is represented in Figure 6.1. Similarly to the API service, the core of the model server is purely functional, while ports and adapters are singletons.
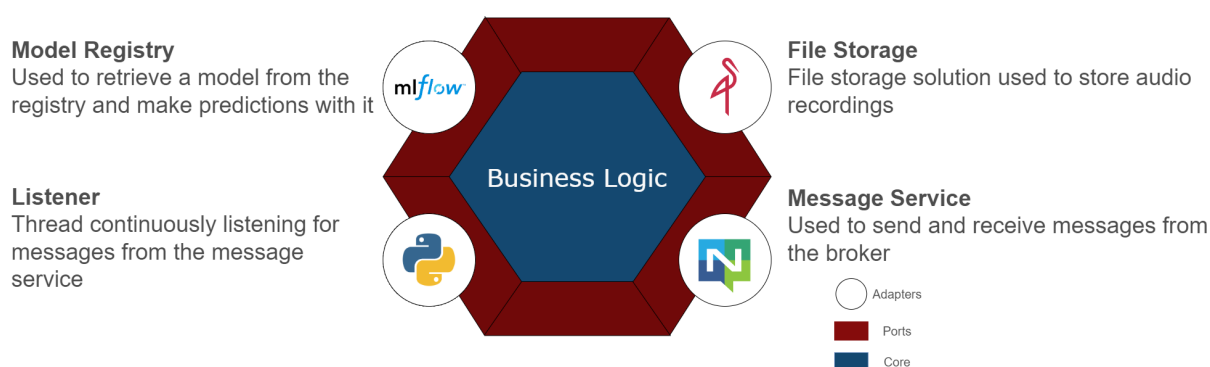

Model Registry
Used to retrieve a model from the registry and make predictions with it

Listener
Thread continuously listening for messages from the message service

File Storage
File storage solution used to store audio recordings

Message Service
Used to send and receive messages from the broker

Business Logic

Adapters
Ports
Core

Figure 6.1: *Architecture of the Model Server.*

6.1 Registry Adapter

The model is recovered from the MLFlow server during model server initialization. More precisely, this retrieval happens inside the MLFlow adapter as presented below:

```python
class MLFlowAdapter:
    def __init__(self, conn_url, model_path):
        logging.info("setting mlflow adapter.")
        self._wait_for_server_connection()
        mlf.set_registry_uri(conn_url)
        logging.info("connected to mlflow server.")
        self._model = mlf.pyfunc.load_model(model_uri=model_path)
        logging.info("model loaded successfully.")

    def predict(
        self, inference: Inference, inference_files: InferenceFiles
    ) -> Tuple[List[float], str]:
        return self._model.predict([inference.dict(),inference_files.dict()])
```

When the adapter is first instantiated, it tries to connect to the MLFlow server. Once the connection is established, the correct model is loaded through a path passed as an environment variable to the adapter. The model instance is stored and will be subsequently used to make predictions.

As cited in Chapter 3, it is important to have a well-defined interface for prediction between the nodel instances and the MLFlow adapter.

The built-in standardized file system for Python models used by MLFlow is called Pyfunc. Along with the file system conventions, PyFunc also provides the abstract class PythonModel that should be inherited by the models in order to store them in the MLFlow server. SPIRA models will then need to implement the following template:

```python
from mlflow.pyfunc import PythonModel


class ModelTemplate(PythonModel):

    def load_context(self, context) -> None:
        """
        This method is called as soon as the model is instantiated.

        The same context will also be available during calls to predict,
        but it may be more efficient to override this method and load
        artifacts from the context at model load time.

        :param context: A :class:`~PythonModelContext` instance containing artifacts
        that the model can use to perform inference.
        """
        pass

    def predict(self, context, model_input) -> Tuple[List[float],str]:
        """
        Evaluates the inference files and metadata and returns a prediction.

        :param context: A :class:`~PythonModelContext` instance containing artifacts
        that the model can use to perform inference.
        :param model_input: A list of two dictionaries, the first containing inference
        metadata and the second containing audio file byte arrays.

        returns: A Tuple where the first element is a list of numbers and the second is
        the diagnosis string ("positive", "negative", "inconclusive").
        """
        pass
```

The `load_context` method is called during model initialization time and it is an optional method for custom initial setups.

The `predict` method is the one used to make inferences with the model. The expected input is a dictionary containing both the inference metadata and the byte arrays of the audio recordings, while the expected output is a Tuple where the first element is the model output, i.e., a list of numbers, and the second element is the diagnosis given by the model.

The adapter will then make use of this template to format the return value of the prediction into the result message for the API.

6.2   Inference Messages

The model server contains only one primary port given by a listener thread. The thread continuously calls the NATS adapter to look for incoming messages at the NATS server in a topic specified among the environment variables. Once a message is received, the core will call the storage port to retrieve the audio files stored by the API in the MinIO server, as described in Figure 6.2.
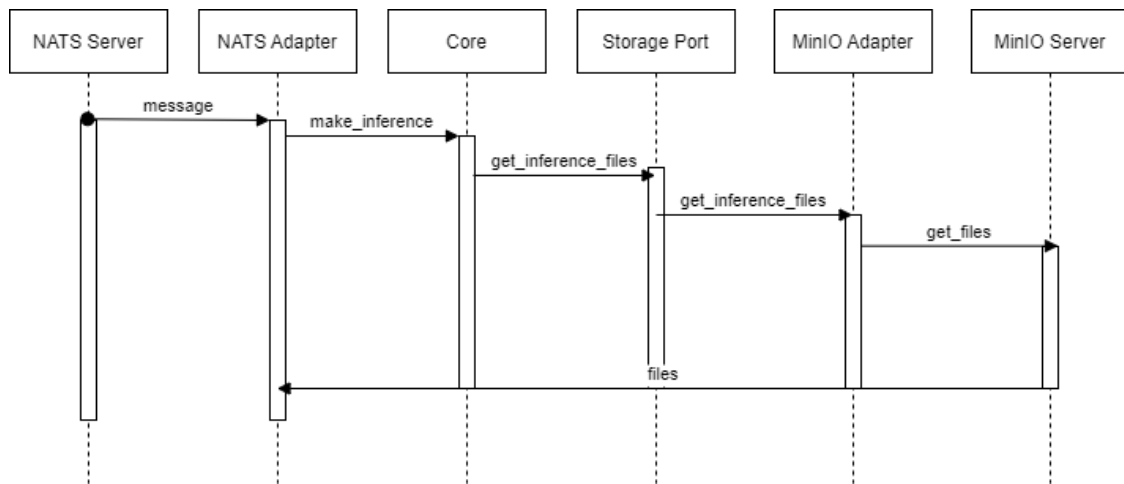
Figure 6.2: *Model server retrieving audio files from MinIO server*

With the audio files and the inference metadata obtained from the received NATS message, the core is able to build the input for the ML model stored in the MLFlow adapter. The sequence of calls follow Figure 6.3. Once the model returns the prediction result, the core encodes it in a NATS message and publishes it in the updates topic being listened by the API.
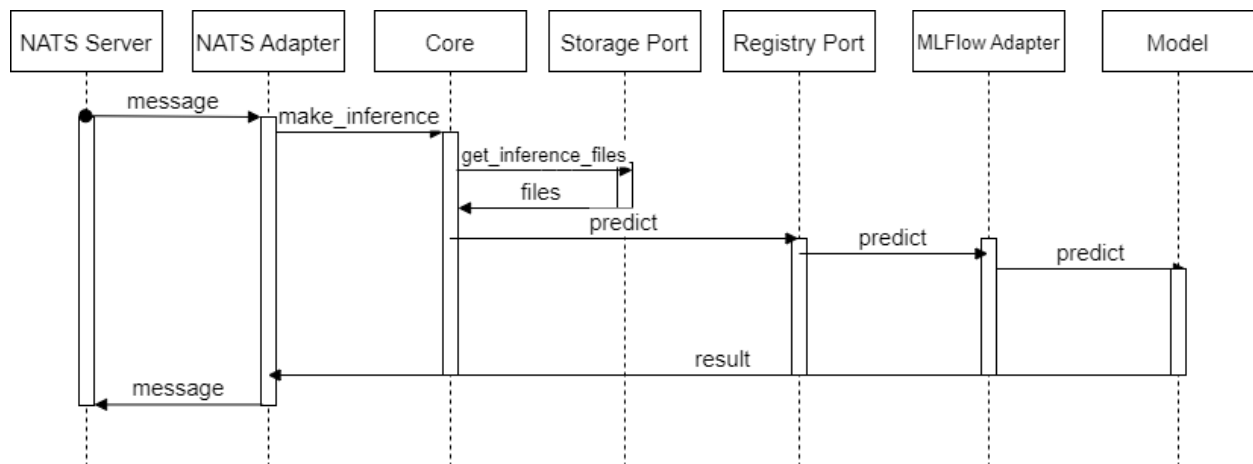

Figure 6.3: *Model server using the model to make predictions*

6.3   ML Pipeline

While the MLFlow allows model creation directly through the UI, there are additional steps that are necessary for the deployment of the model besides registering it in MLFlow server. The deployment of a new model in the inference system can be summarized in following the steps:

- **Model entity insertion in database:** A new model entity needs to be inserted in the database with its name and topic to be used in the message service broker to receive inference request messages.

- **Model registration in MLFlow server:** The model should be loaded to a Python script which connects to the MLFlow server and registers it in the server.

- **Deployment:** Once the model is available in the MLFlow server, the production repository (inf22) needs to be cloned to a new directory and the respective environment variables pointing to the model path in the MLFlow server should be set. The service is then ready to be started.

- **Versioning:** New versions of the same model can also be uploaded to the registry. The staging and production versions of each model can be selected through the MLFlow UI shown in Figure 4.4.

A separate repository was developed to create a pipeline executing the steps described above (mod22). The repository is located at https://github.com/spirabr/Inference-System-ML-Pipeline.

The script fully automates the creation of the model server inside the host machine. The only actions not automatized are the initialization of the service and the versioning of the new model that should still be done through the MLFlow UI.

6.4   Deployment

The development repository (mod22) contains both the implementations of the model server and the MLFlow server. Similarly to the API, a CD pipeline was created with GitHub Actions to automatically upload both images to Docker Hub whenever a new tag is pushed to the repository.
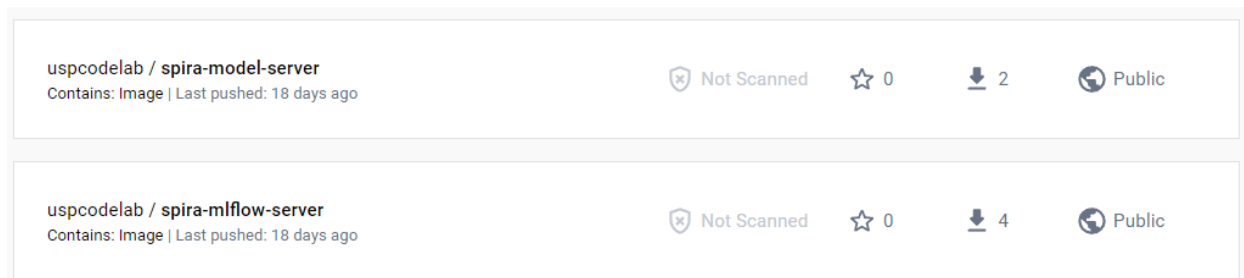

uspcodelab / **spira-model-server**
Contains: Image | Last pushed: 18 days ago   Not Scanned   ☆ 0   ⬇ 2   ⊕ Public

uspcodelab / **spira-mlflow-server**
Contains: Image | Last pushed: 18 days ago   Not Scanned   ☆ 0   ⬇ 4   ⊕ Public

Figure 6.4: *Docker Hub repositories of the model server and the model registry*

As commented in chapter 5, the repository used in production is at (inf22) and it is hosted at https://inspirada.ime.usp.br. The MLFlow server is deployed in the same docker-compose file as the API service, thus they are part of the same systemd unit. The model server on the other hand is deployed as a separate unity so that model servers can be added, updated and deleted without interfering with the rest of the system.

Chapter 7

UI

As shown in section 4.2, a collection PWA was developed inside SPIRA (PWA22). The current design of the App passed through UX reviews in order to guarantee that the layout of the application did not interfere with the reading process of the patients.

To prevent additional bias from being added to the audio recording process of the new inference app, the collection and inference apps are actually different deploy modes of the same project, thus sharing a great part of the UI layout.

The Inference App is hosted inside the Netlify platform (net22) at https://spira-inference.netlify.app. The deployment requires a branch from the project repository which will serve as a production branch.

Netlify then sets an automatic CD pipeline that is updates the application deploy whenever a new push in made to this branch, without the need to use custom Webhooks.

Although the application was developed to be used mainly in mobile platforms, the PWA is also compatible with desktop platforms. Users can also choose between using the application directly from the browser and installing the app in the device.
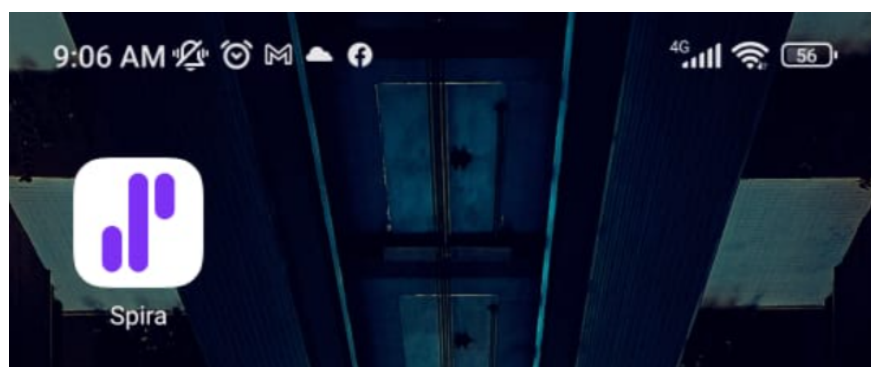

Figure 7.1: *Inference App installed in a mobile device*

To work as intended, the new App required additional features that were not present in the collection PWA.

7.0.1 User Access

While the API provides the mechanisms for authentication, as seen in section 5.3, the UI needs to react accordingly blocking the access of unauthorized users to the application and forbidding users from requesting inference data that they do not own.

The new application contains a new *Sign In* page that users must pass through to access the rest of the UI. The credentials from the page are sent to the API where they are validated. Once the client is authenticated, the application proceeds storing the token locally to send it in the header of
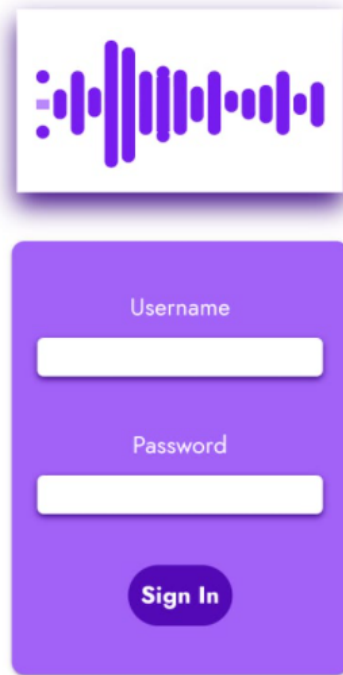
Figure 7.2: *Sign In page in the inference app*

all subsequent requests until it expires or the user signs out of the system, in which cases the users is redirected to the Sign In page. The new page is shown in Figure 7.2.

7.0.2 Inference Form

As cited before, inference and collection processes are similar regarding input generation. Inferences require one additional information regarding the model to be used to make the prediction.

In accordance to that, the new form contains a new select field that retrieves the list of models from the API and the final request now included the ID of the selected model as shown in Figure 7.3.

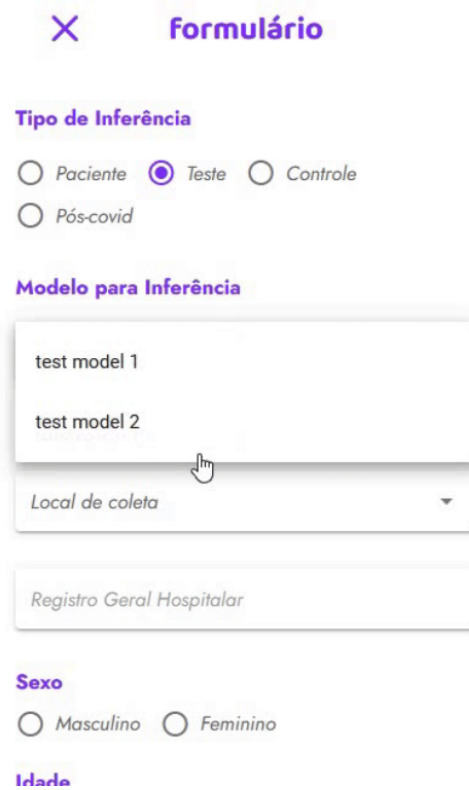Once the form is filed, the user is redirected to make the voice recordings, as shown in Figure 7.4.

At the end of the recording process, the user may send the inference request to the API.

7.0.3 Inference List

After the inference is registered in the back-end, the user may revisit them and check their status through the inferences page. Medical personnel utilize RGH as a unique identifier to the patients. In order to identify an inference, users need to be provided with the following information along with the final diagnosis:

- **Hospital:** Hospital from where the inference was sent. This column is necessary in case inference requestors send requests from different hospitals.

- **Patient RGH:** RGH is the unique patient identifier used by a hospital.

- **Model:** Inferences are intended to be directed to a model. Users should be provided with the model that they chose for a particular inference.

- **Inference Date:** As many inferences may be requested for the same user with the same model, the timestamp of the inference request is also provided for users to identify the inferences they are looking for.

formulário

Tipo de Inferência

○ Paciente   ● Teste   ○ Controle

○ Pós-covid

Modelo para Inferência
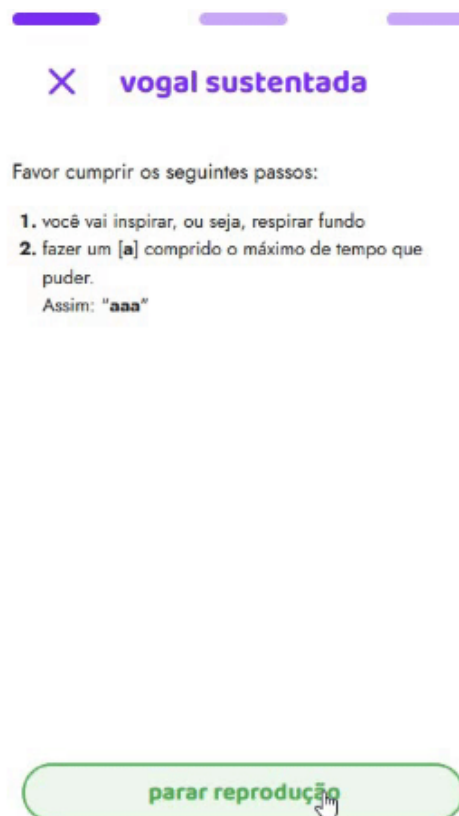
test model 1

test model 2

Local de coleta   ▼

Registro Geral Hospitalar

Sexo

○ Masculino   ○ Feminino

Idade

Figure 7.3: *Inference form page*

vogal sustentada

Favor cumprir os seguintes passos:

1. você vai inspirar, ou seja, respirar fundo
2. fazer um [a] comprido o máximo de tempo que
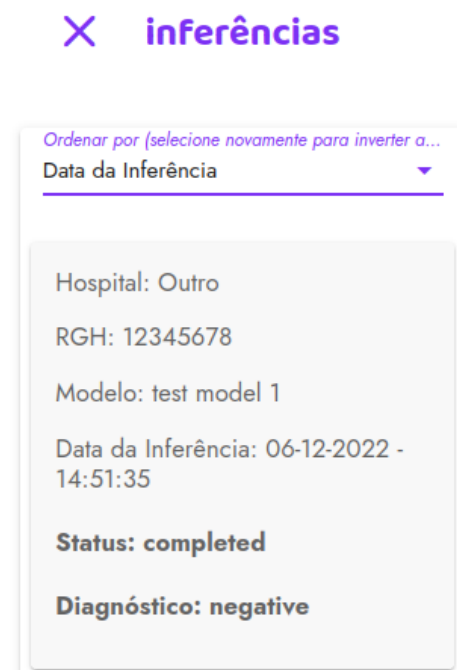   puder.
   Assim: "**aaa**"

parar reprodução

Figure 7.4: *Inference audio recording page*

Figure 7.5: *Inference list page for desktop users*

Figure 7.6: *Inference form page for mobile users*

- **Status:** As cited, users need information regarding the stage of the inference request.

- **Diagnosis:** The diagnosis is needed to show the result of the inference once it is completed or it halts due to an error.

Users can identify an inference with the first four columns and infer the current situation of it with the last two columns. The final desktop layout of the page is presented in Figure 7.5. It is possible to sort the inferences in ascending or descending order with respect to any of the parameters by clicking in the headers of the table.

Due to the width of the mobile screens, a different layout was necessary to show the results for mobile users. The mobile display of the same page is presented in Figure 7.6

Similarly to the desktop version, users are able to sort the inferences by selecting the field at the top of the page.

Chapter 8

Tests

8.1 TDD

Tests were a major step in the development process of the inference system to ensure its correct functioning, and therefore, development techniques that are strongly related to tests were adopted, especially the TDD (Bec03).

TDD or Test Driven Development is a technique that usew short cycles of development and testing to implement the requirements of an application. While not the entire application was developed using TDD, a relevant fraction of it was done through this method.

The development cycle consists in the following steps:

- Write a test that covers a requirement of the system

- Develop the program to pass in the tests

- Refactor the code

- Repeat the process until all requirements are satisfied

This way, tests written by the developer are highly related to the requirements of the system without being biased by the implementation details to achieve them. The technique also encourages developers to write more tests as indicated in (EMT05), since they become a standard procedure for the implementation of new features in the project.

Naturally, a certain overhead is added to the development process, but as indicated in (MP03), the return investment obtained from adopting TDD is highly correlated with Quality Assurance benefits. The current project aims to provide a definitive solution for SPIRA model inferences, therefore it was assumed in this research that the advantages of TDD will outweigh the disadvantages on coding speed in the long term.

8.2 Tests Hierarchy

The tests made can be subdivided in categories, according to their scope:

- Unit Tests: Unit tests are only responsible for testing the behavior of classes and methods.

- Integration Tests: responsible for testing the connection between components.

- System Tests: responsible for testing the entire system flow.

8.2.1   Unit Tests

Unit tests were made with Pytest package. They were used to test the behavior of classes and methods inside the model server and the API service. The three main sections of unit tests are divided in tests for the adapters, ports and the core. In the case of adapters and ports, tests were used to verify the behavior of the classes public methods, whereas for the services located inside the core, the methods themselves were directly tested, as the use cases are purely functional.

The following example is a unit test to check the adapter retrieving a model from the database.

```python
def test_get_model_by_id(database_adapter: MongoAdapter):
    model = database_adapter.get_model_by_id("629f992d45cda830033cf4cd")


    assert model == {
        "_id": ObjectId("629f992d45cda830033cf4cd"),
        "name": "fake_model",
        "publishing_channel": "fake_channel_2",
    }
```

The unit test above is responsible for testing whether the behavior of the database adapter is correct. On one side, the MongoDB client used by the adapter instance is mocked to return false values upon calls to the database. On the other side, the calls made to the adapter, which should come from the ports layer, is also faked by directly calling the method inside the test, further isolating it from the business logic of the microservice. Hence, the test is able to fully isolate the adapter and check if its behavior is correct without the interference of other components.

Likewise for the business logic, according to (Coc), the core should be agnostic to the adapters implementations. Therefore, all the adapters were mocked to isolate business logic from the technologies used in the project.

With each layer being isolated through the mocks, the origin of an error can be more easily identified, as unit tests will point out the layer and class generating the faulty behavior.

8.2.2   Integration Tests

Integration tests were used to check the connection between the containers of the system. Unlike unit tests where connections are mocked, integration tests require the components of the system to be lifted in order to perform them. While Pytest was still used to make the tests, a special tester service was created containing both the microservice source code and the test code, so that after the containers are started, the tests can be executed from within the microservice container. More details on how the tester service was set up are given in section 8.3.

In the same example using database tests, the same test was made *without mocking the mongoDB client* used by the adapter instance. Hence, the test now depends on the connection between the adapter and the database being properly set up, to succeed.

Note that repeating the same test without the mocked database allows one to diagnose the root of an error within the system more efficiently. Failing in the unit test first would imply that the behavior of the method itself is faulty, while passing in the unit test but not in the integration test implies that the adapter behavior is correct, but the connection with the database is not.

8.2.3   System Tests

System tests aim to test the application from end to end. Particularly in this case, the entire project is lifted and requests or messages are artificially sent *from outside* the docker network to test the expected behaviors of the system. Therefore instead of using python scripts, tests were created using bash files that are called from outside the containers. The following example tests the retrieval of inferences, including expected behaviors on malformed, unauthorized and forbidden cases.

```bash
#!/bin/bash
# some details were ommited for the sake of visualization
RESPONSE_STATUS="$(curl \
            --write-out '%{http_code}' --silent --output /dev/null \
               'localhost:3000/v1/users/639686c4ba1604f1387a6c00 \
                /inferences/638f56f70acda5864ee0203a' \
            --header "Authorization: Bearer $TOKEN")"
  .
   .
    .

FORBID_RESPONSE_STATUS="$(curl \
            --write-out '%{http_code}' --silent --output /dev/null \
               'localhost:3000/v1/users/639686c4ba1604f1387a6c01 \
                /inferences/638f56f70acda5864ee0203b'\
            --header "Authorization: Bearer $TOKEN")"

# tests a successfull request
if [ "$RESPONSE_STATUS" -eq 200 ]
  then
      echo 'PASSED';
  else
      echo 'FAILED';
      exit 1;
fi;
# tests a request for inexistent inferences
if [ "$FAILED_RESPONSE_STATUS" -eq 404 ]
  then
      echo 'PASSED';
  else
      echo 'FAILED';
      exit 1;
fi;
# tests a request with no authorization token
if [ "$UNAUTH_RESPONSE_STATUS" -eq 401 ]
  then
      echo 'PASSED';
  else
      echo 'FAILED';
      exit 1;
fi;
# tests a forbidden request
if [ "$FORBID_RESPONSE_STATUS" -eq 403 ]
  then
      echo 'PASSED';
  else
      echo 'FAILED';
      exit 1;
fi;
```

Combined with unit and integration tests, system tests are useful to inspect problems in the
general data flow without the need to concern with the behavior of particular methods, as they
were already tested in unit tests, or the connection between containers inside the docker network,

since they were also checked in integration tests.

They also helpful to preserve the behavior of the system during feature requests, which further ensures that the back-end will behave as expected for outside components such as the UI client.

8.3   Multi-Stage Build

Unit tests of single methods can be done by directly running python test files, however, integration tests that are responsible for testing the connection between the components of the project require a more sophisticated approach.

One way to solve this problem is to use Docker multi-stage build. Multi-Stage build is a feature of Docker used to define various build stages in a single Dockerfile. One is then able to define the stage in which the system should stop building and use the image for diverse purposes. The Dockerfile of the current project contains three different build stages:

- **base:**
  base is responsible for configuring the environment to be used by the next stages. It pulls the python image and installs poetry to manage the dependencies afterwards. It is also responsible for setting the environment variables.

- **dev:**
  dev is the stage that copies the source code and the test code to the image. This image is used to run the tests with the containerized system.

- **prod:**
  prod is the stage that only copies the source code and its necessary dependencies. The image is built to run in production.

For each test sub-routine at all levels of the hierarchy, the whole system is lifted and stopped before the next sub-routine. Particularly, both the API and the model server containers are built with their respective dev images which contain both source and test code. Unit tests and connection tests are then performed from inside these central containers.

In contrast, system tests are performed with the prod image, where no test code is contained in the project and tests are execute from outside the system.

8.4   CI Pipeline

CI Pipeline was implemented in the project with GitHub Actions.

GitHub Actions provides a runner to run custom jobs upon the occurrence of an event. For both development repositories, it was decided that the pipeline would run upon pull requests to the main branch.

The tests execution was automated through Makefiles. They showed to be particularly useful due to the possibility of organizing them in test sections using makefile targets. Each section can then be called separately during development, and a central target was made to run all sections, starting from unit tests, then integration tests and, finally, system tests.

In total, around **170 tests** were created for the model server and the API service combined, including unit, integration and system tests.

Chapter 9

Results

9.1 Milestones

The current research was divided in 4 main stages. chapter 2 was the research for theoretical concepts that were relevant for the planning of the system. chapter 3 focused on the lessons learned from a preliminary version of the inference system. chapter 4 focused on the architectural decisions made based on the acquired concepts and lessons learned. Finally, chapter 5, chapter 6 and section 4.7 showed the implementation process of each component of the final inference system.

From this research, it may be inferred that a complete system was developed to process respiratory insufficiency inferences using trained machine learning models.

The system is fully deployed and medical personnel are now able to use it once the next generation of SPIRA models is created and uploaded to the system.

MLOps practices were followed with the creation of an ML pipeline to deploy new models to the inference system, as it can be seen in Figure 9.1, achieving the two initial motivations: model reproducibility and deployment simplification.
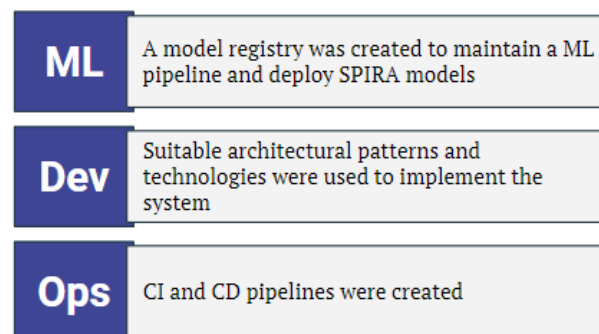
Figure 9.1: *MLOps milestones achieved in the research*

An article containing some of the experiences lived in this project was written and published in ISE 2022.

Figure 9.2: ***Published article link.*** *https://drive.google.com/file/d/1AD1AxlDt6Sp83YW-6cOtSjgU_iNKuuqM/view?usp=sharing*

9.2 Next Steps

9.2.1 Model feedback

While the system offers inference services as initially planned, the inference data stored in the system may be used as feedback input to further train the models and continuously improve their performance.

Nevertheless, this feedback should be carefully planned, as an unplanned feedback pipeline may leave the model exposed to possible attacks to worsen its performance.

9.2.2 NATS Improvements

As the system demand and the number of model servers increase, the load of messages managed by NATS broker is only set to grow. In that case, adding clustering support for NATS would be a great addition for the sake of horizontal scalability of the system.

9.2.3 Users feedback

User feedback regarding the navigability and performance of the system is yet to be collected, as the new SPIRA models being developed still need to be deployed in the system.

Bibliography

[ABC+19] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla et al. Let's encrypt: an automated certificate authority to encrypt the entire web. Em **Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security**, páginas 2473–2487, 2019. 29

[Bec03] Kent Beck. **Test-driven development: by example**. Addison-Wesley Professional, 2003. 38

[Bon16] Jonas Bonér. **Reactive Microservices Architecture**. O'Reilly Media, Inc., 2016. 4, 5

[Cam65] EJM Campbell. Respiratory failure. **British medical journal**, 1(5448):1451, 1965. 1

[CCN+00] Jason W Chien, Russell Ciufo, Ronald Novak, Mary Skowronski, JoAnn Nelson, Albert Coreno e ER McFadden Jr. Uncontrolled oxygen administration and respiratory failure in acute asthma. **Chest**, 117(3):728–733, 2000. 1

[CF20] Jennifer Couzin-Frankel. The mystery of the pandemic's 'happy hypoxia', 2020. 1

[CGC+21] Edresson Casanova, Lucas Gris, Augusto Camargo, Daniel da Silva, Murilo Gazzola, Ester Sabino, Anna Levin, Arnaldo Candido Jr, Sandra Aluisio e Marcelo Finger. Deep learning against COVID-19: Respiratory insufficiency detection in Brazilian Portuguese speech. Em **Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021**, páginas 625–633, Online, Agosto 2021. Association for Computational Linguistics. 1, 8, 9, 12

[Coc] Alistair Cockburn. **Hexagonal architecture**. 6, 21, 39

[DLT+17] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin e Larisa Safina. Microservices: How To Make Your Application Scale. **arXiv e-prints**, página arXiv:1702.07149, Fevereiro 2017. 4

[doc22] Spira images. https://hub.docker.com/u/daitamae, 2022. Accessed: 2022-10-26. 29

[DS94] Joseph V DiCarlo e James M Steven. Respiratory failure in congenital heart disease. **Pediatric Clinics of North America**, 41(3):525–542, 1994. 1

[EGHS16] Christof Ebert, Gorka Gallardo, Josune Hernantes e Nicolas Serrano. Devops. **Ieee Software**, 33(3):94–100, 2016. 3

[EMT05] Hakan Erdogmus, Maurizio Morisio e Marco Torchiano. On the effectiveness of the test-first approach to programming. **IEEE Transactions on software Engineering**, 31(3):226–237, 2005. 38

[FY97] Brian Foote e Joseph Yoder. Big ball of mud. **Pattern languages of program design**, 4:654–692, 1997. 4

[Hul18] Geoff Hulten. **Building Intelligent Systems**. Apress, 2018. 2

[Hut18] Matthew Hutson. Artificial intelligence faces reproducibility crisis, 2018. 3

[inf22] Spira inference system. https://github.com/spirabr/SPIRA-Inference-System, 2022. Accessed: 2022-10-26. 29, 32, 33

[jef08] The onion architecture : part 1. https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/, 2008. Accessed: 2022-12-18. 6

[jwt22] Jwt. https://jwt.io/, 2022. Accessed: 2022-11-29. 24

[Mar96] Robert C Martin. The dependency inversion principle. **C++ Report**, 8(6):61–66, 1996. 6

[MGB+18] Robert C Martin, James Grenning, Simon Brown, Kevlin Henney e Jason Gorman. **Clean architecture: a craftsman's guide to software structure and design**. Number s 31. Prentice Hall, 2018. 6

[min22] Minio. https://min.io/, 2022. Accessed: 2022-11-06. 17

[mlf22] Mlflow tracking server. https://www.mlflow.org/docs/latest/tracking.html, 2022. Accessed: 2022-11-06. 12

[mod22] Spira inference service. https://github.com/spirabr/SPIRA-Inference-Service, 2022. Accessed: 2022-12-29. 33

[MP03] Matthias M Müller e Frank Padberg. About the return on investment of test-driven development. Em **Edser-5 5 th international workshop on economic-driven software engineering research**, página 26, 2003. 38

[NAO+19] Elizamary de Souza Nascimento, Iftekhar Ahmed, Edson Oliveira, Márcio Piedade Palheta, Igor Steinmacher e Tayana Conte. Understanding development process of machine learning systems: Challenges and solutions. Em **2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**, páginas 1–6, 2019. 3

[nat22] Nats. https://nats.io/, 2022. Accessed: 2022-11-06. 15

[net22] Netlify. https://www.netlify.com/, 2022. Accessed: 2022-11-05. 34

[PWA22] Spira pwa app. https://github.com/spirabr/PWA-App/, 2022. Accessed: 2022-10-18. 34

[rea14] Reactive manifesto. https://www.reactivemanifesto.org/, 2014. 4, 5

[reg22] Mlflow server. https://github.com/Fernando-Freire/MLFlow_docker_compose_template, 2022. Accessed: 2022-10-30. 13

[RK03] Charis Roussos e A Koutsoukou. Respiratory failure. **European Respiratory Journal**, 22(47 suppl):3s–14s, 2003. 1

[RTA+21] Ahsab Rahman, Tahani Tabassum, Yusha Araf, Abdullah Al Nahid, Md Ullah, Mohammad Jakir Hosen et al. Silent hypoxia in covid-19: pathomechanism and possible management strategy. **Molecular biology reports**, 48(4):3863–3869, 2021. 1

[Rus15] Alex. Russell. "progressive web apps: Escaping tabs without losing our soul". retrieved june 15, 2015. https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/, 2015. 7

[SLK09]   Meghna R Sebastian, Rakesh Lodha e SK Kabra. Swine origin influenza (swine flu). **The Indian Journal of Pediatrics**, 76(8):833–841, 2009. 1

[SVdB22] Deeks JJ Dinnes J Takwoingi Y Davenport C Leeflang MMG Spijker R Hooft L Emperador D Domen J Tans A Janssens S Wickramasinghe D Lannoy V Horn SR A Struyf, T e A Van den Bruel. Signs and symptoms to determine if a patient presenting in primary care or hospital outpatient settings has covid-19. **Cochrane Database of Systematic Reviews**, (5), 2022. 1

[Teo20]   Jason Teo. Early detection of silent hypoxia in covid-19 pneumonia using smartphone pulse oximetry. **J. Med. Syst.**, 44(8):134, Junho 2020. 1

[TLJ20]   Martin J Tobin, Franco Laghi e Amal Jubran. Why COVID-19 silent hypoxemia is baffling to physicians. **Am. J. Respir. Crit. Care Med.**, 202(3):356–360, Agosto 2020. 1, 2

[TOS⁺20] Mark Treveil, Nicolas Omont, Clément Stenac, Kenji Lefevre, Du Phan, Joachim Zentici, Adrien Lavoillotte, Makoto Miyazaki e Lynn Heidmann. **Introducing MLOps**. O'Reilly Media, 2020. 3