# Making a Pipeline Production-Ready: Challenges and Lessons Learned in the Healthcare Domain

Daniel Angelo Esteves Lawand[1], Lucas Quaresma Medina Lam[1], Roberto Oliveira Bolgheroni[1], Renato Cordeiro Ferreira[1,2,3,4(✉)], Alfredo Goldman[1(✉)], and Marcelo Finger[1(✉)]

[1] Instituto de Matemática e Estatística (IME), University of São Paulo (USP), São Paulo, Brazil
{renatocf,gold,mfinger}@ime.usp.br
[2] Jheronimus Academy of Data Science (JADS), 's-Hertogenbosch, The Netherlands
[3] Technical University of Eindhoven (TUe), Eindhoven, The Netherlands
[4] Tilburg University (TiU), Tilburg, The Netherlands

**Abstract.** Deploying a Machine Learning (ML) training pipeline into production requires good software engineering practices. Unfortunately, the typical data science workflow often leads to code that lacks critical software quality attributes. This experience report investigates this problem in SPIRA, a project whose goal is to create an ML-Enabled System (MLES) to pre-diagnose insufficiency respiratory via speech analysis. This paper presents an overview of the architecture of the MLES, then compares three versions of its Continuous Training subsystem: from a proof of concept Big Ball of Mud (v1), to a design pattern-based Modular Monolith (v2), to a test-driven set of Microservices (v3). Each version improved its overall *extensibility*, *maintainability*, *robustness*, and *resiliency*. The paper shares challenges and lessons learned in this process, offering insights for researchers and practitioners seeking to productionize their pipelines.

**Keywords:** Code Quality · MLOps · Software Architecture · Machine Learning Enabled Systems · Healthcare Domain · Experience Report

## 1 Introduction

In 2020, amidst the COVID-19 pandemic, a multidisciplinary team of researchers from the University of São Paulo (USP) created SPIRA[1]: a project to detect respiratory insufficiency via speech analysis, using Machine Learning (ML) [7].

Since then, the scope of the SPIRA project has evolved to detect respiratory insufficiency of different origins, including many sicknesses that can cause this symptom: smoking side effects, flu, severe asthma, and heart conditions [7].

[1] https://github.com/spirabr.


D. Bianculli et al. (Eds.): ECSA 2025, LNCS 15982, pp. 354–362, 2026.
https://doi.org/10.1007/978-3-032-04403-7_30

To create a tool that could assist physicians, the team proposed to develop the SPIRA ML-Enabled System (MLES). Since 2020, different components have been incrementally developed by bachelor students working with the project.

## 2 The SPIRA ML-Enabled System

This section describes the architecture of the SPIRA ML-Enabled System, as illustrated by Fig. 1. Sections 2.1 to 2.6 showcase its six subsystems according to the reference architecture extended by Ferreira *et al.* [5].

### 2.1 Data Collection

Creating an ML model that can detect respiratory insufficiency via voice is a supervised machine learning classification problem [4]. As such, it requires *labeled data*. The SPIRA team supports a Data Collection App (1) provided to volunteer data collectors to collect voices inside hospitals, where people with respiratory insufficiency may be found. The Data Collection App (1) sends data to a Data Collection API (2), which in turn stores audio in an Audio Key-Value Database (A), and saves patients' info in a Document Database (B).

After each data collection, data collectors register a unique identifier for each participant. This ID can be cross-referenced with data shared by partner hospitals via 3rd-party Hospital APIs (3). This way, researchers can create a Label Store (C) that provides a *ground truth* for training models.

### 2.2 Continuous Training

Casanova *et al.* proposed a Deep Learning (DL) architecture based on a Convolutional Neural Network (CNN) to detect respiratory insufficiency from audio signals [4]. As more data becomes available as *ground truth*, there is potential to retrain the model and improve its accuracy [10].

The proof-of-concept shared by Casanova *et al.* [4] can be divided into two components: a Feature Engineering Pipeline (I), to process audio signals for the training algorithm; and a Training Pipeline (II), to train and validate the CNN.

The development of these two pipelines is an opportunity to introduce standard machine learning design patterns [5,12], such as a Feature Store (D), a Metadata Store (E), and a Model Registry (I). For the SPIRA project, the goal is to use MLFlow to make the two latter roles. On the other hand, for the features, the proposal is to use a simple key-value database like MinIO.

### 2.3 Development

As more data becomes available as *ground truth*, there is potential to redesign the architecture of the model used by SPIRA to improve its accuracy [10]. To make experimentation easy, it is desirable to provide a standardized environment for data scientists, compatible with production environments [21].

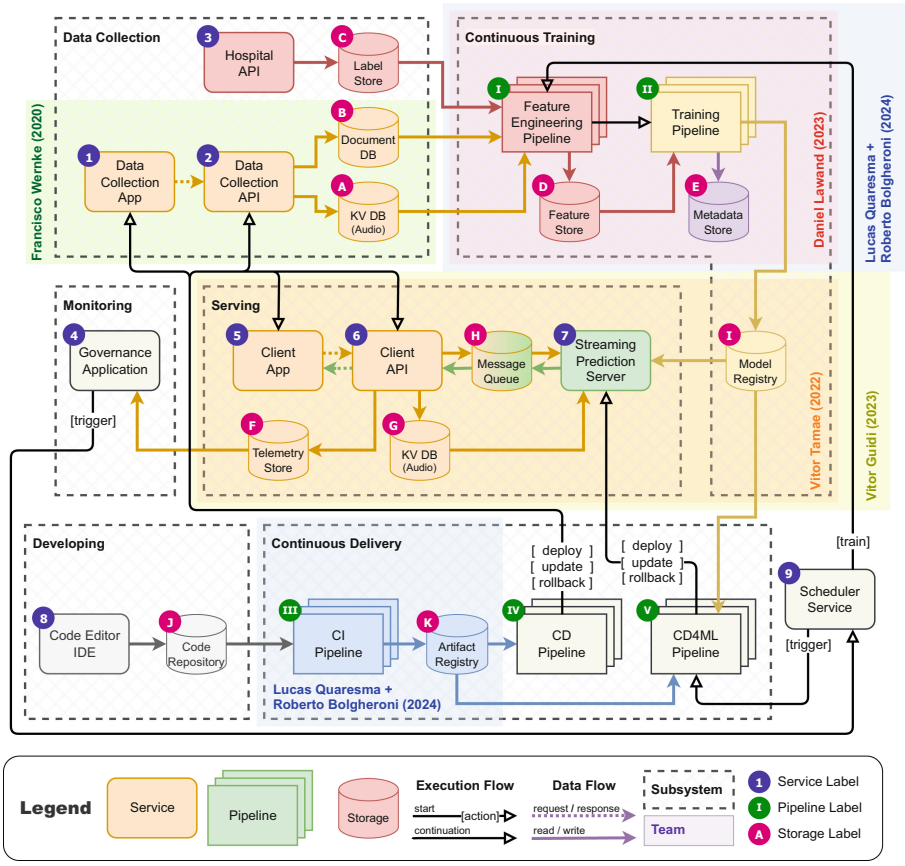


**Fig. 1.** *Architecture of the SPIRA ML-Enabled System.* The architecture is described with the same notation of the reference architecture described in Ferreira *et al.* [5]. Rectangles represent **services**, which execute continuously. Stacked rectangles represent **pipelines**, which execute a task on demand. Lastly, cylinders represent **data storage**, which may be databases of any type. Components are connected by arrows. Black arrows with a hollow tip illustrate the **execution flow**. They start and end in a component. Labeled arrows represent the trigger that starts a workflow, whereas unlabeled arrows represent the continuation of an existing workflow. Colored arrows with a filled tip illustrate the **data flow**. They appear in two types: solid arrows going to and from a data storage represent write and read operations, respectively; dotted arrows represent a sync or async request-response communication between components. Components are colored according to the data they produce: raw data ■, source code ■, executable artifacts ■, ML-specific data ■, ML models ■, ML training metadata ■, ML model predictions ■, and ML model metrics ■. The remaining standalone components □ orchestrate the execution of others. Components are also grouped into **subsystems**, with their background colored according to the **students** responsible for their development. Numbers ●, roman numerals ● and letters ● are used as labels throughout Sect. 2.

Currently, all SPIRA source code is open source in a Code Repository (J) at GitHub. Following the principles of *Infrastructure as Code (IaC)* [17], the SPIRA organization provides standard configurations for popular development tools used by data scientists and machine learning engineers. In particular, this includes a basic setup for Code Editors (8) (such as VSCode) or IDEs (8) (such as PyCharm), but can also include other utility scripts helpful for the developers.

### 2.4 Continuous Delivery

To support the creation of new versions of the MLES – after retraining or redesigning a model – automation is essential to make deployment seamless.

Taking advantage of GitHub, a Continuous Integration Pipeline (III) uses GitHub Actions to generate Docker containers for components, which are then deployed into the GitHub Container Registry, the de facto Artifact Store (K) for SPIRA. Conversely, the Continuous Delivery Pipeline (IV) and Continuous Delivery for Machine Learning Pipeline (V) use infrastructure configuration to deploy and rollback services in a Kubernetes-managed cluster.

### 2.5 Serving

Similarly to the Data Collection App (1), the SPIRA Client App (5) is meant to be used inside hospitals. However, it has a different goal: to provide (indirect) access to the SPIRA model, pre-diagnosing respiratory insufficiency via speech analysis. Therefore, its data collection is more critical.

Hospitals can have poor, unreliable internet reception. Multiple audios can be sent in a single burst whenever a user gets internet access. To prevent data loss, the Client App (5) must store all data collected locally before sending it to its corresponding Client API (6). Moreover, to prevent data inconsistency, the Client API (6) must validate all data received with the corresponding Client App (5) before storing it in its Audio Key-Value Database (G).

Deep Learning models such as the one proposed by Casanova *et al.* often require accelerator hardware (GPUs) to run efficiently [10]. Since executing the model is a costly operation, handling arbitrarily bursts of request is difficult.

To avoid this bottleneck, the SPIRA ML model can be executed in a separate Streaming Prediction Service (7), decoupled from the Client API (6). A Message Queue (H) stores prediction requests and corresponding results. Thanks to this separation, the Streaming Prediction Service (7) may be run in machines with GPUs, whereas other components are deployed in cheaper hardware.

### 2.6 Monitoring

Once the SPIRA MLES reaches production, it needs to be maintained in operation. The Telemetry Store (F) stores useful logs and data about its usage, while the Governance Application (4) summarizes statistics of its working status based on them. Combined, these two components help the SPIRA team to decide to retrain the ML model, or consider redesigning the ML model, or update and redeploy a component by triggering the Scheduler Service (9).

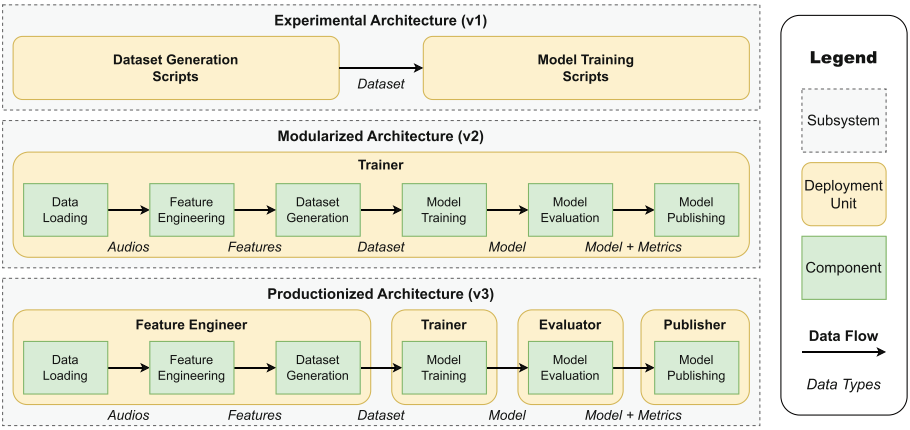


**Fig. 2.** *Incremental Development of the SPIRA* Continuous Training *Subsystem.* Dashed boxes ▢ represent the versions of the subsystem. Rounded boxes ▢ represent deployment units. Squared boxes ▢ represent components. Black arrows with a filled tip illustrate the **data flow**, while the labels in *italic* represent the **data types**.

## 3 Incremental Implementation

The SPIRA MLES has been incrementally developed since 2021. This paper focuses on the Continuous Training subsystem, whose development process is summarized in Fig. 2. For challenges and lessons learned while developing the Data Collection and Serving, subsystems, please refer to our paper *"SPIRA: Building an Intelligent System for Respiratory Insufficiency Detection"* [6].

## 4 Experimental Architecture (v1)

The first version of the Continuous Training pipeline was created as a proof-of-concept to showcase the SPIRA model. The code was published as part of the reproduction package by Casanova *et al.* in 2020 [4].

The package has two main sets of scripts: one to create datasets (available in the `scripts` folder), and one to train the CNN model (available in the `models` folder). The code presents an unplanned, organically developed structure, as it often occurs in the experimentation phase by data scientists [12,21].

Figure 3 (left) illustrates an example of the Misplaced Responsibility bad smell [15] in the code, evidence of high coupling and low cohesion in the code. In this way, this architecture can be classified as a Big Ball of Mud [8].

## 5 Modularized Architecture (v2)

The second version of the Continuous Training pipeline was created during the bachelor thesis of Daniel Lawand [14]. Its code is open source at GitHub.

( v1 architecture )

```
random.seed(c.train_config["seed"])
torch.manual_seed(c.train_config["seed"])
torch.cuda.manual_seed(c.train_config["seed"])
np.random.seed(c.train_config["seed"])
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

self.c = c
self.ap = ap
self.train = train
self.test = test
self.test_insert_noise = test_insert_noise
self.num_test_additive_noise = num_test_additive_noise
self.num_test_specaug = num_test_specaug
self.dataset_csv = \
  c.dataset["train_csv"] if train else c.dataset["eval_csv"]

assert os.path.isfile(self.dataset_csv), \
  "Test or Train CSV file don't exists! Fix it in config.json")

accepted_tc = [ 'overlapping', 'padding', 'one_window' ]
assert self.c.dataset['temporal_control'] in accepted_tc), \
  "You cannot use the padding_with_max_length option with the \
   split_wav_using_overlapping option, disable one of them !!")

self.control_class = c.dataset['control_class']
self.patient_class = c.dataset['patient_class']

self.dataset_list = \
  pd.read_csv(self.dataset_csv, sep=',') \
  .replace({'?': -1}) \
  .replace({'negative': self.control_class}, regex=True) \
  .replace({'positive': self.patient_class}, regex=True) \
  .values
```

( v2 architecture )

```
# Setup

config_path = ValidPath.from_str("/app/spira/spira.json")
config = load_config(config_path)

operation_mode = OperationMode.TRAIN
randomizer = initialize_random(config, operation_mode)

# Data Loading

patients_paths = read_valid_paths_from_csv(config.patients_csv)
controls_paths = read_valid_paths_from_csv(config.controls_csv)
noises_paths = read_valid_paths_from_csv(config.noises_csv)

patients_inputs = Audios.load(
  patients_paths, config.audio, config.dataset
)
controls_inputs = Audios.load(
  controls_paths, config.audio, config.dataset
)
noises = Audios.load(noises_paths, config)
  noises_path, config.audio, config.dataset
)

# Feature Engineering

audio_processor = create_audio_processor(config.audio)

patient_feature_transformer = create_audio_feature_transformer(
    randomizer, audio_processor, config, noises,
)
control_feature_transformer = create_audio_feature_transformer(
    randomizer, audio_processor, config, noises,
)
```

**Fig. 3.** *Example of improving modularization between* v1 *and* v2 *architectures.* The v1 snippet (left) shows a Misplaced Responsibility bad smell [15] at the class `Dataset`. Lines 1–7 handle random number generation. Lines 8–16 assign values to attributes. Lines 18–19 and 22–24 handle assertions. Line 30 handles data loading. The v2 snippet (right) shows the application of design patterns at the module `pipeline`. Line 15–23 handles data loading using the `Audio` Adapter. Line 27 builds an `audio_processor` via a Chain of Responsibility pattern. Line 29–36 build `feature_transformer`s via the Strategy pattern. Improving the modularization makes the intention of the code more explicit: allow multiple experiments depending on the configuration.

This version applied multiple design patterns [9] to modularize the code. Some examples include: Chain of Responsibility, to dynamically choose the pipeline steps to be executed; Strategy, to dynamically choose techniques for data preprocessing, feature engineering, and model evaluation; and Template Method, to reuse generic code snippets. In another level of abstraction, the business logic was decoupled from external dependencies by using the Ports and Adapters architectural pattern [16], using Dependency Injection to connect different layers of the application.

Figure 3 (right) illustrates the use of design patterns. This reimplementation made the code more *maintainable* and *extensible* [19]. The v2 architecture can be classified as a Modular Monolith [20].

## 6 Productionized Architecture (v3)

The third version of the Continuous Training pipeline was created during the bachelor thesis of Lucas Quaresma and Roberto Bolgheroni [13]. Its code is open source at GitHub.

This version expands on the v2 architecture by splitting the single monolithic application into multiple deployment units, as described by the Workflow Pipeline ML design pattern [12]. The goal was to enable executing the pipeline with a Scheduler Service (9). In this way, if a failure occurs in one stage of the pipeline, it does not require re-executing it from the start, a costly operation.

Testability was also a key priority in this version. It was only possible because the v2 architecture made clearer the expected behaviors from the v1 architecture. Following testing practices for microservices [20], the business logic received unit tests via *Test-Driven Development* (TDD) [1], while the Ports and Adapters received integration tests around external dependencies [16].

This reimplementation made the code more *robust* and *resilient* [19]. The v3 architecture follows a Microservices architectural style [18].

## 7 Challenges

The Experimental Architecture (v1) was built following a typical CRISP-DM process [3,21]. The goal was to create a proof-of-concept model. Serving it, i.e., going into production, was the last step of the workflow.

This development process brought a series of drawbacks for the Continuous Training subsystem's architectural characteristics, in particular its *extensibility*, *maintainability*, *robustness*, and *resiliency*. As a consequence, there were two main challenges to migrating the subsystem toward production.

*Separation of Concerns.* The v1 architecture had many examples of Misplaced Responsibility bad smell [15], which affected its overall *maintainability* and *extensibility* [19]. Building the v2 architecture became an exercise in *Software Archeology* [11]: it was reimplemented from the v1 code by carefully reading it line by line to decipher its intentions, applying design patterns to decrease coupling and increase cohesion in its abstractions.

*Automated Testing.* The v2 architecture had improved *modularity*, but it still lacked *robustness* and *resiliency* [19]. Building the v3 architecture became an exercise in *Test-Driven Development* (TDD) [1]: it was reimplemented from v2 code by carefully interpreting intentions around its design, creating automated tests to document behaviors that should be maintained for the long term.

## 8 Lessons Learned

The development of MLES is inherently difficult, and developing them without adequate planning makes their complexity even greater. As a consequence, there were two lessons learned that could have improved the migration of the Continuous Training subsystems toward production.

*Collaboration between Data Scientists and ML Engineers.* Machine Learning Engineering (MLE) is a new subarea of software engineering whose goal if to help to productionize ML models [3,21]. This discipline helps the data scientist to think about how it will fit into an MLES. By bringing these two roles together since the beginning, it is easier to make code *maintainable* and *extensible*.

*Testing as a First-Class Concern.* Creating automated tests is recognized as a good practice in software engineering [2]. This discipline helps developers to think about the long-term maintenance of a system. By designing tests and validation since the beginning, it is easier to make the code *robust* and *resilient*.

## 9 Conclusion

This paper addressed the incremental development of SPIRA, focusing on its Continuous Training subsystem. Its architecture evolved in three stages: from a proof of concept Big Ball of Mud (v1), to a design pattern-based Modular Monolith (v2), to a test-driven set of Microservices (v3). Each step helped the subsystem's *extensibility*, *maintainability*, *robustness*, and *resiliency*.

By learning from this experience, similar projects may employ the above lessons learned to avoid similar challenges, thus reaching production sooner.

**Acknowledgements..** This research was funded by FAPESP (São Paulo Research Foundation), grant number 2023/00488-5.

## References

1. Beck, K.: Test Driven Development: By Example, 1st edn. Addison-Wesley Professional (2002)
2. Beck, K., Andres, C.: Extreme Programming Explained, 2nd edn. Addison-Wesley Professional (2004)
3. Burkov, A.: Machine Learning Engineering, 1st edn. True Positive Inc. (2020)
4. Casanova, E., et al.: Deep learning against covid-19: respiratory insufficiency detection in Brazilian Portuguese speech. In: Findings of the Association for Computational Linguistics, pp. 625–633 (2021)
5. Ferreira, R.C.: A Metrics-Oriented Architectural Model to Characterize Complexity on Machine Learning-Enabled Systems. Proceedings - 2025 IEEE/ACM 4th International Conference on AI Engineering - Software Engineering for AI, CAIN 2025 (2025)
6. Ferreira, R.C., Gomes, D., Tamae, V., Wernke, F., Goldman, A.: SPIRA: building an intelligent system for respiratory insufficiency detection. In; Workshop Brasileiro de Engenharia de Software Inteligente (ISE), vol. 1, pp. 19–22 (2022)
7. Finger, M., et al.: Detecting respiratory insufficiency by voice analysis: the SPIRA project. In: Acoustic Communication: An Interdisciplinary Approach (2021)
8. Foote, B., Yoder, J.: Big ball of mud. In: PLoP '97 / EutoPLoP '97. Monticello (1999)

9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, 1st edn. Addison-Wesley Professional (1994)
10. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning, 1st edn. The MIT Press (2016)
11. Hunt, A., Thomas, D.: Software archaeology. IEEE Software **19**(2) (2002)
12. Lakshmanan, V., Robinson, S., Munn, M.: Machine Learning Design Patterns, 1st edn. O'Reilly Media (2020)
13. Lam, L.Q.M., Bolgheroni, R.O., Ferreira, R.C., Goldman, A.: Productionizing SPIRA's Model Trainer System. Ph.D. thesis, University of São Paulo, São Paulo (2024)
14. Lawand, D.A.E., Ferreira, R.C., Goldman, A.: Enabling MLOps in the SPIRA Training Pipeline. Ph.D. thesis, University of São Paulo, São Paulo (2023)
15. Martin, R.C.: Clean Code, 1 edn. Pearson (2008)
16. Martin, R.C.: Clean Architecture, 1st edn. Pearson (2017)
17. Morris, K.: Infrastructure as Code, 2nd edn. O'Reilly Media (2025)
18. Newman, S.: Building Microservices, 2nd edn.. O'Reilly Media (2021)
19. Richards, M., Ford, N.: Fundamentals of Software Architecture, 1st edn. O'Reilly Media (2020)
20. Richardson, C.: Microservices Patterns, 1st edn. Manning Publications (2018)
21. Wilson, B.: Machine Learning Engineering in Action, 1st edn. Manning Publications (2022)