UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ARCHITECTURAL REDESIGN AND EVALUATION OF AN OPEN SOURCE MLOPS PLATFORM: A CASE STUDY OF APACHE MARVIN-AI

LUCAS CARDOSO SILVA

ORIENTADOR: PROF. DR. DIEGO FURTADO SILVA

São Carlos – SP

Junho/2021

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ARCHITECTURAL REDESIGN AND EVALUATION OF AN OPEN SOURCE MLOPS PLATFORM: A CASE STUDY OF APACHE MARVIN-AI

LUCAS CARDOSO SILVA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software
Orientador: Prof. Dr. Diego Furtado Silva

São Carlos – SP

Junho/2021

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Dissertação de Mestrado do candidato Lucas Cardoso Silva, realizada em 01/07/2021.

Comissão Julgadora:

Prof. Dr. Diego Furtado Silva (UFSCar)

Prof. Dr. Valter Vieira de Camargo (UFSCar)

Profa. Dra. Elisa Yumi Nakagawa (USP)

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

Aos meus pais.

AGRADECIMENTOS

Agradeço primeiramente a minha família, que em todos os momentos me ajudou a superar as dificuldades durante meu percurso, me proporcionando abrigo, conselhos e sabedoria para que eu pudesse dar o meu melhor.

Ao meu orientador, Prof. Dr. Diego Furtado Silva, e ao Prof. Dr. Daniel Lucrédio, que através de suas vastas experiências e competências, me auxiliaram em todas as dúvidas e questionamentos que apresentei. Sem a assistência e dedicação que tiveram para transmitir seus conhecimentos na realização da pesquisa, não seria possível o correto desenvolvimento deste trabalho.

Aos membros da banca examinadora, Profa. Dra. Elisa Yumi Nakagawa e Prof. Dr. Valter Vieira de Camargo, que gentilmente aceitaram participar e colaborar com este projeto de pesquisa. Suas contribuições serão utilizadas prontamente a fim de aprimorar as ideias e atingir os resultados esperados.

À empresa B2W Digital, parceira desta pesquisa, que me proporcionou a possibilidade de trabalhar dentro de um projeto real em paralelo com a realização do meu projeto de mestrado.

À Profª. Dra. Helena de Medeiros Caseli, coordenadora do projeto Marvin em parceria com a B2W Digital, que além de prestar assistência e apoio durante o projeto, também realizou diversas contribuições para a minha pesquisa.

Aos companheiros que entraram no mestrado e no projeto da B2W Digital na mesma época que eu: Bruno Silva Sette, Fernando Rezende Zagatti e Lucas Nildaimon dos Santos Silva. Amigos que acrescentaram muito conhecimento durante nossas discussões e trocas de experiências. Sem eles, nada disso seria possível.

Ao corpo docente do Departamento de Computação da Universidade Federal de São Carlos pelo alto nível de ensino e condução das atividades. Tenho certeza que cada dia foi uma experiência de aprendizado única.

Por fim, a todos os amigos e amigas que conheci no Departamento de Computação pelo convívio, amizade e apoio. Também, a todos os profissionais e acadêmicos que contribuem compartilhando seus conhecimentos e experiências, direta ou indiretamente, permitindo a realização desta pesquisa, o meu sincero agradecimento.

Were it not for the leaping and twinkling of the soul, man would rot away in his greatest passion, idleness.

C.G. Jung

RESUMO

Aprendizado de máquina é um termo vinculado à ciência de dados, uma área multidisciplinar que engloba conhecimentos da ciência da computação, matemática e experiência em um domínio. Dada esta natureza multidisciplinar, uma grande variedade de desafios se apresenta para seus praticantes, visto que uma vasta gama de habilidades é necessária para treinar modelos e colocá-los em produção. Parte desses desafios pode ser resolvida com a ajuda de ferramentas e plataformas de aprendizado de máquina. Nesse contexto, a plataforma de aprendizado de máquina de código aberto Apache Marvin-AI oferece uma maneira padronizada de desenvolver e colocar modelos de aprendizado de máquina em produção. Embora o Apache Marvin-AI tenha muito a oferecer para iniciantes e cientistas de dados que não possuem as habilidades de engenharia de software para lidar com os problemas mencionados anteriormente, faltam recursos desejados por usuários mais avançados. Para resolver este problema, foi realizada a evolução e avaliação arquitetural do Marvin-AI. O processo foi guiado por uma versão simplificada do ATAM (Architecture Tradeoff Analysis Method), que foi adaptada para funcionar em um ambiente de desenvolvimento distribuído de código aberto. O resultado do processo foi avaliado de quatro formas distintas: (i) análise estática de código-fonte; (ii) feedback das partes interessadas; (iii) a análise de taxonomia para avaliar a maturidade das soluções desenvolvidas; e (iv) uma avaliação das novas features de monitoramento. No geral, o processo de concepção, implementação e avaliação da nova arquitetura foi considerado bem-sucedido por todas as quatro avaliações independentes, e as lições aprendidas constituem importantes contribuições deste trabalho.

Keywords: Avaliação arquitetural, MLOps, ATAM

ABSTRACT

Machine learning is a term linked to data science, a multidisciplinary area that encompasses knowledge of computer science, mathematics, and domain experience. Given this multidisciplinary nature, a wide variety of challenges are presented to its practitioners, as a wide range of skills is required to train models and put them into production. Part of these challenges can be solved with the help of machine learning tools and platforms. In this context, the Apache Marvin-AI is an open-source machine learning platform that offers a standardized way to develop and put machine learning models into production. While Apache Marvin-AI has a lot to offer for novices and data scientists who do not have the software engineering skills to deal with the aforementioned issues, it lacks features desired by more advanced users. To solve this problem, an architectural evolution and evaluation was carried out. The process was guided by a simplified version of ATAM (Architecture Tradeoff Analysis Method), adapted to work on a distributed open-source development environment. The results of this process were analyzed in four different ways: (i) source code static analysis; (ii) feedback from stakeholders; (iii) taxonomy analysis to assess the maturity of the developed solutions; and (iv) an assessment of the new monitoring features. Overall, the process of designing, implementing, and evaluating the new architecture was deemed successful by all four independent evaluations, and the lessons learned are important contributions from this work.

Keywords: Architecture evaluation, MLOps, ATAM

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

ATAM – *Architecture Tradeoff Analysis Method*

CI/CD – *Continuous Integration and Continuous Delivery*

DAG – *Directed Acyclic Graph*

DASFE – *Data Acquisition, Selection, Feedback and Evaluation*

LLOC – *Logical Lines of Code*

MLOps – *Machine Learning Operations*

ML – *Machine Learning*

MLaaS – *Machine learning as a service*

RFC – *Request for Comments*

SEI – *Software Engineering Institute*

TFX – *Tensorflow Extended*

CONTENTS

Chapter 1

INTRODUCTION

Machine learning is a term linked to data science, a multidisciplinary subject that encompasses knowledge from computer science, mathematics and experience in a domain (FINZER, 2013). Given this multidisciplinary nature, a wide variety of challenges is presented. Part of these challenges can be addressed with the help of machine learning tools and platforms such as Tensorflow[1], Scikit-learn[2] and Pytorch[3]. They normally incorporate the latest advancements in terms of algorithms for machine learning tasks, allowing non-computer scientists to build ML solutions. However, there are many challenges that fall outside the scope of these tools, such as:

- **Dynamism of real problems:** frequently, machine learning models present a good performance in the training examples, but are not able to generalize well when applied in real problems due to the constant reorganization of the data (SCULLEY et al., 2015).
- **Reproducibility and scalability:** most of the problems found in large institutions are concerned with scalability issues due to the large flow of information that they have to deal with every day, making it difficult to reproduce the algorithms (SCULLEY et al., 2015; LIN; RYABOY, 2013).
- **Complexity:** the majority of problems presented by companies are complex and increasingly difficult since they encompass several areas that use different types of techniques (LIN; RYABOY, 2013; SCULLEY et al., 2015).

[1] https://tensorflow.org
[2] https://scikit-learn.org/
[3] https://pytorch.org/

- **Loss of efficiency:** Over time, machine learning models end up losing efficiency due to data versatility, demanding constant supervision (SCULLEY et al., 2015; GHANTA et al., 2019).

These are not machine learning problems, but they represent important issues that surround a machine learning project. They represent things that must be dealt with when the machine learning solution is put into a production environment. In this scenario, through certain practices, Machine Learning Operations (MLOps) emerged for stipulating favorable habits and environments for machine learning tools, bringing quick benefits for businesses and offering reliability during production.

MLOps is a term derived from DevOps (the combination of "development" and "operations"), which has the goal of assisting in a better coordination between software development and operation. The focus of the DevOps movement is to make development teams more aware of the main issues of software operation, such as deployment, logging, monitoring, and incident handling, treating these tasks as "first-class" citizens from the point of view of requirements (BASS; WEBER; ZHU, 2015). Similarly, MLOps gives data scientists more responsibility regarding the integration of their solutions into a production environment.

In this context, the machine learning platform Apache Marvin-AI[4] offers a standardized way to develop and bring ML models into production. Most of Marvin-AI's design decisions aim at smoothing the learning curve of the MLOps process. Although Apache Marvin-AI has a lot to offer to beginners and data scientists who do not have the software engineering skills to deal with the previously mentioned problems, it lacks features desired by more advanced users. Advanced functionalities such as reliable batch scheduling, cluster execution management and metadata persistence were lacking proper support at the platform.

Aiming to solve these problems, an architectural evolution effort was carried out. The evolution was partly guided by ATAM (Architecture Tradeoff Analysis Method), a well-known method for architecture evaluation (KAZMAN; KLEIN; CLEMENTS, 2000). But because Marvin-AI is currently incubated at the Apache Software Foundation, there were some limitations imposed by the open-source development model. Therefore we had to simplify ATAM to make it work mainly through Apache Marvin-AI's developers mailing list, with some occasional synchronous meetings and open source techniques such as RFCs.

The process led to six main design decisions that were incorporated into a new architecture for Marvin-AI:

[4] https://marvin.apache.org/

- Containerization of Marvin-AI's solutions to allow for better support for logging, monitoring, scaling and crash recovery;
- Conversion of some of Marvin-AI's code into a daemon[5] structure to allow remote development;
- Integration with Tensorflow Extended (TFX), a powerful open source MLOps platform that incorporates some of Marvin-AI's missing features (cluster execution, metadata persistence, anomaly checking, among others);
- Support for an easy configuration of Apache Airflow to perform batch scheduling tasks;
- Improvement on CLI capabilities in terms of code generation; and
- The development of a benchmark suite for monitoring ML solutions.

The details behind these improvements, together with the lessons we learned from conducting an architectural evolution in an open source environment using a simplified ATAM, are the main contributions of this research. We also present the results of four evaluations that demonstrate the superiority of the new architecture, thus confirming that, from the perspective of the desired output, the simplified ATAM was successful.

In the remainder of this essay, we first discuss related work (Chapter 2), focusing on the main findings related to MLOps and the main quality attributes expected from a platform such as Marvin-AI. Next we describe Marvin-AI's original and redesigned architecture and the method (simplified ATAM) used in this process (Chapter 3). We also present four evaluations (chapter 4) using the developers' feedback, static source code metrics, a taxonomy evaluation to assess the maturity level that can be achieved by Marvin-AI's solution and an evaluation with Marvin-AI's new monitoring features. The results demonstrate that the new architecture fulfills the needed quality attributes and point out some limitations. We conclude the essay (Chapter 5) by discussing how this research led to more robust capabilities desired by more advanced users, without sacrificing Marvin AI's mission of being helpful to those users without much experience in MLOps. We also discuss future directions for the next Marvin-AI releases

[5] A daemon is a computer program that executes as a background process instead of being attached to a user interface.

Chapter 2

BACKGROUND AND RELATED WORK

2.1 Review description

The literature review was made following the next directives:

- Classic related MLOps papers, as Sculley et al. (2015). These papers are cited by a large amount of other related work.
- Papers from MLOps related conferences, as USENIX OPML.
- Papers that presents MLOps platforms, methods or techniques. For reach the max amount of related papers, we to achieve as many related papers as possible, systematize the review using snowballing.
- Grey literature high quality work, as there are not many MLOps work in academic literature.

Changing the architecture of a software system without the support of a robust method such as ATAM can lead to problems. Behnamghader et al. (2017) identified that architectural degradation is commonly caused by ad-hoc changes made during software development. Based on two empirical studies with open source software from the Apache Software Foundation, the researchers concluded that: (i) some intracomponent architectural changes may subtly hinder the architecture, because they are small and difficult to evaluate; and (ii) drastic architectural changes may occur even close to releases, when stability should be a priority, thus contributing to architectural degradation. Alenezi and Khellah (2015) also investigated architectural degradation at package level, using an instability metric (MARTIN, 2002) to help identify the issues. After evaluating two open source projects, they observed a tendency of instability increase over time.

A solid methodology can also help to produce a good architectural analysis and documentation, which can be beneficial for an open source project such as Apache Marvin-AI. Kazman et al. (2016) present an analysis where the researchers developed an architecture document for a very large open source project. Making the documentation available in the Internet produced two interesting effects: (i) more users started to join the community; and (ii) casual committers started to become more involved in core development, as the knowledge became less centralized. The study concludes that improvements on architecture documentation has a key role at internal graduation of new developers, adoption rate and knowledge decentralization. We expect to obtain similar benefits in Apache Marvin-AI.

Barbacci et al. (2003) report the application of the ATAM method in a avionics system developed by Rockwell Collins in Cedar Rapids, Iowa U.S., for the U.S. Army Special Operations helicopters called CAAS. This was a new system at the time, replacing third part proprietary and closed source systems, that was designed to be extensible and provide a common look-and-feel across similar systems of its kind. Phase one of the evaluation was conducted with 12 stakeholders, an above the average mark as mentioned at the technical report, with: (i) five of then being architects and engineers from the contractor, (ii) two members of the organization that had the task to implement the Special Operations Aviation systems changes; and five members of the 160th Special Operations Aviation Regiment, representing the user community. In this phase, six scenarios were analyzed. Phase two was conducted with 15 stakeholders, with member from user community, governmental agency and contractor workers. In this phase they added 9 scenarios to the evaluation, for a total of 15. Overall, the analysis was considered successful, as the main quality attributes were evaluated and the architecture has succeeded in fulfilling them. The partitioning of the architecture components was the main point of discussion as this design decision influences the maintainability and extensibility of the software. The evaluation concluded that the architecture was robust regarding the ability to add third-party functionalities and to support new hardware. The evaluation has also raised concerns about requirements that were out of scope and declared as being too hard to implement in the architecture. These issues were brought to the program office and handled by other means.

In the machine learning area, there are not many academic research papers related to software architecture. However, there are many studies and discussions regarding important concerns and known issues that represent architectural quality attributes. These go beyond algorithm performance. As stated by Flaounas (2017), although it may seem that putting a model into production consumes only 10% of the effort, practice has shown that it might consume up to 90% if proper provisions are not taken.

Nguyen et al. (2019) presented a survey to describe the machine learning and deep learning frameworks and libraries that tended to be used in the state-of-the-practice. The study compiled various open source frameworks and libraries, developed by academia or industry, that use special paradigms such as map-reduce. An analysis of positive and negative points and a review of the tools was performed (NGUYEN et al., 2019). In the end, the article discusses, among other things, the trend of using Python as the main language for data analysis and the hegemony of large corporations in developing open source frameworks and libraries for deep learning (NGUYEN et al., 2019).

Lim et al. (2019) specify an operational lifecycle scheme for machine learning projects in the manufacturing industry to increase productivity. According to the authors, using machine learning in production means more than just training and running models. Their proposed MLOps lifecycle, which is called CruX, deals with large volumes of raw data, data cleaning and data labeling, which cannot be easily carried out in a single server (LIM et al., 2019). At the same time, the use of off-premise services is not adequate due to confidentiality. Their scheme fulfills these requirements through edge computing, which allows MLOps tasks be carried out without compromising the production.

Sridhar et al. (2018) discuss model governance, which is an important requirement for Machine Learning in production. Model governance involves proper management and documentation of the entire pipeline to help solving problems such as: knowing the provenience/lineage of an algorithm (knowing precisely which combination of events, data, algorithms, versions and platforms led to a particular result); reproducibility (the ability to recreate a previous result); auditing and conformance (to fulfill regulations); reuse (avoiding rework); scale and heterogeneity (governance must work in real environments, with large amounts of data and with different technologies/languages); multiple uses of metadata (different people might use governance with different goals).

Governance was also considered as a major requirement for continuous validation for data analytics systems by Staples, Zhu and Grundy (2016). In this paper, the authors discuss the concept of continuous usage validation, which is particularly difficult for systems that employ Machine Learning, since models are normally treated as “black-box” components, difficult to be inspected. Knowing exactly who created the model, where the data came from, how was the training carried out, among other details, is crucial for this task. Having to deal with secrecy and ethics also requires good model governance.

Lwakatare et al. (2019) present an empirical investigation of a set of software engineering challenges for Machine Learning systems, according to different areas: data collection, model

creation, training, evaluation and deployment. The challenges are also categorized according to the level of maturity in Machine Learning inside the company. In this study, the authors identified five maturity levels, the first representing those companies that are just starting to try Machine Learning, and the last representing companies where Machine Learning is well established, to the point of being completely autonomous (the system monitors and controls itself). In each level, the challenges are different. For example, hidden feedback loops and undeclared model consumers are known challenges, but are not an issue for companies in the first level. Only in the fourth level, when multiple ML models are used together, one serving as input to another, these issues start to appear.

In a similar study, Serban et al. (2020) identified a set of 29 engineering practices important for machine learning, some of these being related to MLOps. The authors categorized these practices into 6 classes: Data, Training, Coding, Deployment, Team and Governance. The authors have also mapped which of these practices produce the different effects, identifying possible actions to be made by teams based on their needs. For example, teams that need agility should focus on increasing the adoption of practices that can lead to this effect.

Both of these studies (LWAKATARE et al., 2019; SERBAN et al., 2020) highlight the fact that different companies, teams and users have different needs. This is also an important motivation of this work, which is to make Marvin more flexible to support a wider range of user needs.

2.2 Quality Attributes

We surveyed the literature in search for MLOps requirements, both from academia and industry. The goal was to establish a set of architectural quality attributes for an MLOps platform, as required by ATAM (KAZMAN; KLEIN; CLEMENTS, 2000). Some of the identified requirements refer to more abstract quality properties, such as “Usability” and “Robustness”, while others have a more functional meaning, such as “Monitoring” and “Batch execution”. But because all of them have impact on the architecture, in this work we are going to call them quality attributes. Next we present the results of this survey.

Usability. As Finzer (2013) points out, the knowledge set of data scientists is diverse, and many of them do not have the necessary software engineering skills to easily navigate into the hassles of software development, deployment and operation. A good MLOps platform should make life easier for these professionals, by hiding as many technical details as possible and allowing data scientists to focus on models and exploring the data (HERMANN; BALSO, 2017).

CI/CD support. The platform's architecture must be able to execute all components of the pipeline, generate artifacts and update the production version in a automated manner, similarly to the continuous integration and delivery practices in DevOps. Testing data dependencies and models with different metrics and from different perspectives in each pipeline execution is crucial to establish a fully independent pipeline, not relying on human interference to check data quality or algorithm configuration (SCULLEY et al., 2015; LWAKATARE et al., 2019).

Robustness. The platform should deal with adverse conditions, such as sudden growth of requests, abnormal data formatting and partial failure. Ideally, such unexpected conditions should not interfere in availability and stability (BAYLOR et al., 2017; SCULLEY et al., 2015; LIN; RYABOY, 2013).

High availability. A machine learning solution has to be available the majority of the time. When performing a version rollback or updating the artifacts in production, the MLOps platform must do it in a transparent way, without any interruption (BAYLOR et al., 2017; **??**).

Standardization. Standards help to reduce the memory burden of a software developer, by making things similar across different scenarios. In MLOps, standards should be followed through the entire pipeline: components should communicate using a generic interface, common to all tasks. This allows developers to make their own components to replace ones that are not suitable to certain tasks (BAYLOR et al., 2017; SCULLEY et al., 2015). Standards also facilitate model governance, by making it easier to inspect and understand the different models and artifacts.

Component reuse. All of the produced ML components must be made independent and standardized enough to be used in other pipelines, as needed. Unregistered dependencies should be eliminated, and a pipeline should not interfere in the performance of another (SCULLEY et al., 2015; LIN; RYABOY, 2013).

High performance. Some ML tasks such as data processing and hyperparameter optimization are very hardware consuming. To reduce the time spent, the pipeline has to be optimized. Executing parallel tasks whenever possible is a good way to make better use of existing hardware (ZHOU et al., 2019).

Prototype to deployment. A good MLOps platform should offer some way for the data scientist to easily run their experiments and prototypes at scale, avoiding the prototype smell described by Sculley et al. (2015), and help in the deployment of their solution. This helps the data scientists to keep their solutions inside the standardized environment all the time, avoiding many ML and software engineering problems (FINZER, 2013).

Monitoring. An MLOps end-to-end pipeline has to be monitored in each of its steps. For computational resources, monitoring is important to detect systems overloading, an indicative that performance is been affected by hardware boundaries (SILVA et al., 2020; ANDREWS et al., 2015; PROMETHEUS, 2021). Monitoring model metrics in production environment helps to detect staleness and detachment between model predictions and the real world. Data monitoring is important to detect anomalies in the dataset distribution and schema during the pipeline execution (SCULLEY et al., 2015).

Extensibility. An MLOps platform should allow its components to be modified or replaced by new ones created through an API. Extensibility is fundamental in ML because of the large variety of tasks, algorithms, techniques and raw data formats (BAYLOR et al., 2017).

Batch execution. When training models, some tasks can take a lot of time and computational resources, like: hyperparameter tuning, neural architecture search and data wrangling. These tasks rely on the platform's ability to support batch executions, allowing task tracking, proper debugging and testing (HERMANN; BALSO, 2017; BAYLOR et al., 2017).

This is not a complete list, as different platforms may introduce specific quality attributes. Also, some of these may be more important than others, depending on the platform's goal. This is why a prioritization, such as the one prescribed in ATAM (KAZMAN; KLEIN; CLEMENTS, 2000), is recommended.

Chapter 3

METHODOLOGY

Apache Marvin-AI is a standardized framework for developing and deploying machine learning solutions. It was designed to be simple enough for data scientists without much technical background in MLOps. Next we describe the old architecture, and how it fulfilled some of the quality attributes described in the previous section.

3.1 The old architecture

First, it is important to stress that Marvin-AI is not a replacement for other tools, such as Apache Spark[1], Spark MLlib[2] and Tensorflow Keras[3]. It was designed to make their usage easier, by introducing a design pattern and standardized components that make it easier to move an ML solution into the production environment.

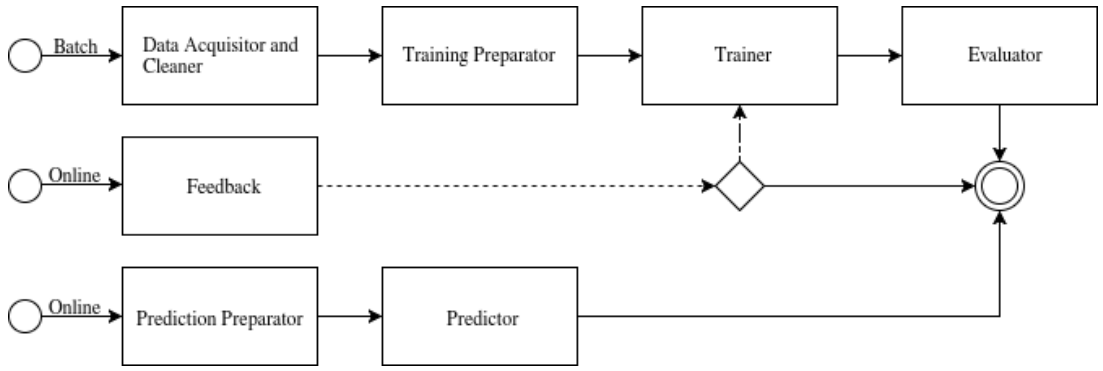

Figure 3.1: DASFE design pattern (APACHE, 2021b)

[1] https://spark.apache.org/
[2] https://spark.apache.org/mllib/
[3] https://keras.io/

DASFE (Data Acquisition, Selection, Feedback and Evaluation) is Marvin-AI's design pattern for the development of machine learning as a service (MLaaS) applications. It provides a framework for a distributed pipeline based on components that can perform actions in batch or online. As shows Figure 3.1, the components that run in batch are: Data Acquisitor and Cleaner, Training Preparator, Trainer and Evaluator. They perform all the common steps in an ML pipeline: initial data acquisition and cleaning, data transformation, training and model evaluation (**??**APACHE, 2021b).

Each batch job produces an intermediate artifact for communication with subsequent components. The Trainer component, for example, must load the transformed dataset produced by the Training Preparator component and persist the trained model for future use.

The online components are: Prediction Preparator, Predictor and Feedback. The Prediction Preparator receives new data for making predictions through an endpoint. This component is responsible for carrying out the necessary transformations for preparing the input for the model trained in the batch steps. After the Prediction Preparator executes, the Predictor component runs, returning the model's prediction through the API provided by Apache Marvin-AI. The feedback action is optional and may allow the user to send feedback data through a specific endpoint, aiming at a possible interpretation and model re-training.

Marvin-AI does not require DASFE to be used from the beginning. The developer can start prototyping and experimenting in a standard Jupyter Notebook[4], without having to worry about DASFE tasks or a deployment-ready code structure. When the developer is happy with the results and wants to move the code to production, the developer must mark the code to identify which parts of it correspond to each DASFE component. Marvin-AI provides specific tags, in a custom Jupyter plugin, for this marking. Because DASFE represents a typical ML pipeline, its components should be easily identifiable in most prototypes. Communication between tasks must also be standardized in this step: the developer must refactor the code that persists data to produce the intermediate artifacts required by each DASFE task.

Once the code is fully tagged and the communication between the tasks is standardized, the code is exported to a class structure that is prepared to be deployed as a gRPC server[5]. Each task will be wrapped as a gRPC service, so that it can be executed independently through gRPC calls.

Another important feature of Apache Marvin-AI is the Engine Executor. The task of the Engine Executor is to connect as a client to the gRPC servers and initiate a series of HTTP

[4]https://jupyter.org/
[5]https://grpc.io/

endpoints to allow the management of the actions. Batch actions can be executed to generate new artifacts, reload the artifacts used in their execution and perform a health check routine. Online actions can also perform health checks and reload artifacts used in their execution. In addition, the *"prediction"* endpoint performs the Prediction Preparator and Predictor actions sequentially, receiving new data as input and returning the model prediction. The *"feedback"* endpoint receives the client's message and returns a receipt verification message (**??**).

Before the Engine Executor and gRPC servers initialization, the developer must execute the *dryrun* command, which is part of Marvin's CLI, to perform all actions, some integration tests between the steps of the pipeline and produce the initial artifacts. These initial artifacts are loaded by the actions when the gRPC servers initializes. When receiving a message for the execution of any action, the Engine Executor generates a protocol number. This number must be informed whenever a message is sent for some action to reload (APACHE, 2021b).

Apache Marvin-AI also has a CLI command for the creation of a basic code structure. This structure, referred as engine, contains a skeleton of DASFE's actions classes, configuration files and unit testing structure. When creating the engine, a GIT repository is initialized at the engine's root directory for code versioning. Other features include commands for data synchronization and the use of TDD (Test Driven Development) (**??**).

The deployment of a Marvin engine can happen in multiple ways, depending on the user's needs, since each action is an independent microservice whose interface with external applications and services occurs through the Engine Executor. The Trainer action, for example, may require the use of a GPU for training the model. In this case, the Trainer action gRPC server can be initialized on a environment that contains the necessary hardware resources. Similarly, the Training Preparator action typically presents high memory consumption, therefore it must run on a machine or container that has enough of this resource (SILVA et al., 2020).

In this old architecture, Apache Marvin-AI has proven to be a powerful tool. The DASFE design pattern is simple enough to shorten the learning curve for a data scientist who wants to put his machine learning models into production. In particular, the following quality attributes were adequately supported by Marvin-AI:

- Usability - Marvin-AI was designed to be easy to use, based on the premise that most data scientists are familiar with Jupyter notebooks. DASFE is also a simple pattern, requiring little effort to adapt a typical pipeline to its components. The CLI has only a few commands and there is a simple web interface for running the tasks.

- CI/CD support - Marvin-AI's engine executor is able to execute pipeline components, reload artifacts and perform health checks. All these tasks are executed asynchronously and the produced artifacts can be stored remotely in Amazon S3, Azure or HDFS. Marvin-AI also allows users to perform unit and integration tests between the pipeline stages trough its CLI. For tests that need a complex environment, such as A/B tests, manual configuration is required from the user.

- Standardization - the DASFE design pattern is easy to learn and keeps the engine code standardized. During engine project generation, all the basic configuration files needed for development are generated, in an organized and human-readable format.

- Prototype to deployment - Marvin-AI has a customized Jupyter Notebook extension that allows the developer to add markups to the code and export it into DASFE classes, making it production-ready. The developer also has several CLI tools to test and deploy this code.

The other identified quality attributes, however, were lacking proper support by the tool:

- Robustness: Marvin-AI has support for storing artifacts in external services such as HDFS and Amazon S3, therefore providing good support for data storage and retrieval. Also, Marvin-AI's solutions are exposed as a set of GRPC services and an HTTP server, which can be replicated to support high demand peaks. However, this had to be done by hand, externally;

- High availability: proper support for failure recovery was not addressed by Marvin-AI. While the Engine Executor had a simple health check mechanism, automatic service restart or more complete resource monitoring was not present and had to be configured by hand;

- Monitoring: the logging present in Marvin-AI was not enough to allow proper information necessary for debugging and operation;

- Component reuse: Marvin-AI exposes its tasks as GRPC services and HTTP endpoints, therefore it is possible to reuse them in other pipelines. But using other components as part of a Marvin-AI solution requires some adaptation in the code;

- High performance: Marvin-AI's solutions can make use of parallel execution, but the developer has to implement these features by hand. The DASFE design pattern doesn't allow multiple classes for the same action. When training multiple models, for example, the developer has to combine them into a single instance of the Trainer component.

This analysis was not performed in an exhaustive or systematic way, but it indicated that better support for these quality attributes was needed. It also indicated that an architectural evolution had to be carried out, as these are not simple features that can be simply added in new releases. We will discuss the architectural changes in the next subsection.

3.2 The new architecture

When designing the new architecture, the development team decided to reuse as many existing technologies as possible. This led to six main design decisions, as described next.

3.2.1 Containerization of Marvin-AI's solutions

The first architectural decision was to replace Python's Virtualenv[6] with Docker[7] containers, since the container isolates the dependencies not only from the Python language libraries but also from the operating system. The use of containers allows the deployment strategy to be less dependent on the bare hardware environment. Containers also make it easier to perform logging, monitoring, scaling and crash recovery, which were identified as important architectural quality attributes for an MLOps platform. Finally, the adoption of Docker containers also facilitates the execution of algorithms and prototyping solutions remotely and in a distributed way through pipeline orchestration at container level. This is useful, as most of the big data tasks have hardware requirements that are rarely met by data scientists' workstations (SCULLEY et al., 2015), making working in managed cloud environments a common scenario.

Containerization was achieved by changing Marvin-AI's code generation templates to produce the configuration files required by Docker. But to make the remote development scenario possible, another change had to be implemented as well. Marvin-AI's command line interface (CLI) had some of its code moved into a daemon process. This was necessary because the container must maintain some background process while waiting for instructions from a remote client.

The daemon was implemented using gRPC protocol, to maintain consistency with Marvin's online and batch actions, which have not been modified in the new architecture. Each function performed by the daemon is implemented in the protocol, including: (1) integration and unit tests; (2) gRPC servers setup for actions passed by parameter; (3) iterative development note-

[6]https://pypi.org/project/virtualenv/
[7]https://www.docker.com/

book setup. The specification of the task to be performed and its parameters are sent via gRPC from the CLI to the daemon through the service port.

The connection is insecure, but Marvin-AI's CLI has tools to protect it by connecting to the daemon port through an SSH[8] tunnel. The SSH connection is authenticated through a pair of public and private keys generated by Marvin-AI's CLI when creating the engine. By doing so we guarantee that the insecure port isn't exposed by the firewall, but only the container's SSH port and the Jupyter Notebook port that has its own authentication implemented by default.

In addition to the components of the daemon and DASFE actions communication, some tooling implementations are present in the daemon code. In addition to the serializers, which have not changed, there is also a wrapper for the standardized use of the Tensorflow Extended (TFX)[9] framework, which is another design decision, described later in this essay.

The serving strategy in the new architecture is similar to the previous one. The Engine Executor component connects itself to the gRPC actions servers, the main difference being that, as it is using Docker containers by default, it is easier to deploy the Marvin actions and the Engine Executor composing a microservice architecture in a cluster management system like Kubernetes[10]. Resource monitoring and logging are also easier as they can benefit from Docker's tools (SILVA et al., 2020; DOCKER, 2021).

The introduction of Docker containers introduced better support for the following quality attributes:

- Robustness: containerization facilitates the recovery from unexpected crashes, as containers can be easily replaced and supports fast failure recognition, as the container stops when the main process dies.
- High availability: a containerized solution can be more easily replicated during demand spikes and the load can be balanced between the replicas, making the solution available even under high demand conditions.
- High performance: the remote development, allowed at the containerized daemon, can provide access to better hardware, including GPUs, essential to neural networks parallelism.

[8] Secure Shell (SSH) is a secure network protocol to perform encrypted connections in a insecure network. A SSH Tunnel allows the user's system to link a remote system port into a local port trough the SSH port.

[9] https://www.tensorflow.org/tfx/

[10] https://kubernetes.io/

- Prototype to deployment: the changes allow the developer to use the prototyping tools, such as notebooks, in a containerized environment, thus guaranteeing that this same environment will be used in the deployment.

3.2.2 Integration with Tensorflow Extended

Another change implemented in the new architecture was the incorporation of a wrapper for TFX - TensorFlow eXtended. TFX is a TensorFlow-based general-purpose machine learning platform implemented at Google to facilitate the process of putting models in production, bringing benefits in terms of standardization, robustness, monitoring, among others (BAYLOR et al., 2017). There are two different ways to use TFX, in terms of how the code is written and executed: interactive execution and DAG orchestration. If the developer opts for interactive execution, there is not much difficulty, as it is done in a Jupyter notebook or Colab (TFX, 2021). This does not take full advantage of TFX's serving capabilities, but it might be adequate for scenarios where offline execution is all that is needed.

For more demanding scenarios, with batch and online actions being executed constantly and under high demand, TFX has the option to create a directed acyclic graph (DAG) for a more complex component pipeline, including persistence settings for metadata and parameters. The DAG can be executed in a pipeline orchestrator (such as Kubeflow Pipelines - KFP[11] or Apache Airflow[12]) (TFX, 2021). But properly setting up such orchestration requires considerable effort and a specific knowledge that isn't often part of the data scientist's skill set.

This is a problem that Marvin-AI already solved, for a simpler case. In Marvin-AI, the code is written in a Jupyter notebook, but is deployed as separate components according to DASFE. Each component has its own endpoints for remote execution, making them ready to use by applications or end users. Compared to TFX, orchestration is simple and fixed. Components have to be executed all at once, or one by one, using a simple web interface. But in the end, the Marvin-AI approach hides the complexities of orchestration behind a simple notebook and web interface.

By including an interface between TFX and Marvin-AI, opting for the standardization of the pipeline along the lines of DASFE over the extensibility of TFX, we can abstract a good part of the verbosity of the configuration required to build an Airflow or KFP production pipeline DAG in TFX. The result is the combination of TFX's benefits with Marvin-AI's simplicity.

[11] https://www.kubeflow.org/
[12] https://airflow.apache.org/

The proposed class for the interface between the two frameworks is `MarvinTfxContext`, whose structure its entirely based on TFX's `InteractiveContext` class. `MarvinTfxContext` class runs TFX components in a directory and records the executions metadata in the Machine Learning MetaData library (MLMD)[13]. In this case, there is not much difference between the proposed class and the `InteractiveContext` offered by default in the TFX library.

The usage is exemplified in Listing 3.1. TFX's "gen" components are used normally (lines 2-3 and 4-6), as in standalone TFX. But the remaining code is embedded into `MarvinTfxContext` (line 1), so that the developer has less configuration to do. The class constructor's default definition, without passing any parameter, is enough to persist the metadata in a SQLite[14] database file and use a directory to persist the artifacts, both inside Marvin-AI's data folder. The developer can use different configurations to define, for example, another database manager to perform metadata persistence.

Listing 3.1: MarvinTfxContext Usage

```
1 context = MarvinTfxContext()
2 example_gen = CsvExampleGen(input=external_input(_data_root))
3 context.run(example_gen)
4 statistics_gen = StatisticsGen(
5                       examples=example_gen.outputs['examples'])
6 context.run(statistics_gen)
```

When running the solution interactively, persistence will be done in a temporary directory. However, when executing the solution orchestrated by Marvin-AI, a root directory is generated with a unique identifier that persists these artifacts. The metadata, in the interactive way, is also persisted in a SQLite database on a temporary directory, while in the orchestrated solution we will have a standard configuration for the database (SQLite persisted in Marvin's standard directory structure), but it may be configured according to the developer's needs using another database management systems (DBMS) or file paths.

The developer can export the code in the DASFE standard to an Apache Airflow DAG, containing the steps: Acquisitor, Training Preparator, Trainer and Evaluator. The DAG code is made available to the developer, who can modify it or use it as it was generated. Orchestration with Marvin-AI also has the benefit of allowing code outside the TFX components to be executed between steps in the pipeline.

[13] https://github.com/google/ml-metadata

[14] SQLite is a small, fast, self-contained, high-reliability, full-featured SQL database engine implementation (SQLITE...,).

As represented in Listing 3.2, the TFX implementation at Marvin-AI also features a class for accessing TFX artifacts directly from the metadata store, that can be defined passing an instance of `MarvinTfxContext` (line 1). The access to the persistent artifacts is represented at lines 3 and 4, when the methods `get_examples()` and `get_schema()` are called. The `TfxArtifacts` instance is returning the examples and schema persisted in other actions. This allows the unification of the interactive and orchestration modes, since in the latter the variables are reset after each action, as they are executed in functions within different class scopes. The other lines follow the default flow of Marvin TFX integration, as with the definition of TFX pipeline components and the context instance executing them.

Listing 3.2: TfxArtifacts class usage

```
1 artifacts = TfxArtifacts(context)
2 transform = Transform(
3     examples=artifacts.get_examples(),
4     schema=artifacts.get_schema(),
5     module_file=os.path.abspath(_taxi_transform_module_file))
6 context.run(transform)
```

TFX has a Docker image that is used by orchestrators to run their components independently, offering standardized input and output (TFX, 2021; BAYLOR et al., 2017). Although this pattern is similar to the one used by the Marvin-AI daemon, since it can run each action independently passing them through a parameter through a gRPC call, the way in which each image acts is different.

That said, we can produce a new daemon image based on the TFX image base containing the dependencies to both. While TFX has its own execution mode for Docker, which consists of sending a component to the image using gRPC with standardized input and output, we use the execution method provided in `InteractiveContext` (the `in_process_component_launcher` function), which executes the components as an internal process of the system, making the container not limited to a TFX component and adapting to Marvin-AI's actions (TFX, 2021).

Another possibility for the user is to build an Apache Beam[15] DAG within Marvin's actions. In the solution illustrated in Listing 3.3, we can see it applying the same workflow as standalone TFX (lines 1-10). At line 12, a list with the components is defined and, in the line 16, a method from the context named `run_beam_dag()` is called to build an Apache Beam DAG and run it.

[15] Apache Beam is an unified software model for defining both batch and streaming data-parallel processing pipelines (APACHE, 2021a)

As the `ExampleValidator` (line 1) and `Transform` (line 7) components can be executed in parallel, this strategy is applied to the Beam DAG (TFX, 2021).

Listing 3.3: Beam DAG in MarvinTfxContext usage

```
1  example_validator = ExampleValidator(
2      statistics=statistics_gen.outputs['statistics'],
3      schema=schema_gen.outputs['schema'])
4
5  _taxi_transform_module_file = 'taxi_transform.py'
6
7  transform = Transform(
8      examples=example_gen.outputs['examples'],
9      schema=schema_gen.outputs['schema'],
10     module_file=os.path.abspath(_taxi_transform_module_file))
11
12 pipeline = [
13     example_validator,
14     transform
15 ]
16 context.run_beam_dag(pipeline)
```

As mentioned in the previous section, serving with Docker allows for easier deployment, cluster management, monitoring and logging. When the TFX wrapper is being used, a new option is now available. As Tensorflow has its own serving API, the models can be deployed using this serving component instead of Marvin-AI's Engine Executor. When using TFX on Marvin-AI, the user has a different workflow than the default one. The solution ends in the model evaluation because Tensorflow Serving does not require any user code in its setup.

In summary, the Marvin-AI/TFX integration has the potential to enhance the support for the following quality attributes:

- Robustness: the serving component of Tensorflow is designed for production environments. The component provides an easy way to serve Tensorflow models, being flexible and extensible for different solutions and data.
- High availabilty: the serving component can be easily configured and replicated trough containers. The load can be distributed to several containers, using Kubernetes or another load balancer solution.

- Monitoring: TFX has the MLMD component that deals with metadata gathering and persistence. Trough this, Marvin-AI has a much more complete solution to this process than the standard one keeping track of all experiments execution steps.

- Component reuse: as TFX has a more standardized way to define pipeline components, using Tensorflow's toolkit to process data, train, evaluate models and gRPC[16] protocol for comunication between components, component reuse is better achieved with this integration.

- High performance: Tensorflow has several parallelism strategies implemented and TFX allows its use at maximum. In addition, Apache Beam allows even data acquisition tasks to execute in multiple workers.

- Standardization: this quality attribute has improved in two aspects: (i) TFX's component pattern defines a general pipeline structure for the developer to follow; and (ii) the DASFE pattern is still supported, even in conjunction with TFX code pattern.

3.2.3 Support for an easy configuration of Apache Airflow

Apache Airflow is a tool to build and monitor workflows in a simplified and extensible way, allowing the generation of dynamic DAGs and extension of its workflow components, known as operators.

To facilitate the adoption of Airflow in Marvin-AI, a new CLI feature was incorporated. The objective of this feature is to make the access to the engine executor easier through Airflow's bash operator. The CLI command is able to start batch and online actions, perform health checks and assert that some action has finished its execution. All these functionalities can be easily performed in bash operator.

The DAG code generated by Marvin-AI is available to the user to make adaptations such as more complex tasks for data acquisition or some customized artifact management, but it is also ready to be used as it is.

The improved orchestration in this new architecture can provide better support for the following quality attributes:

- Usability: Apache Airflow has a web user interface that improves the visualization of batch pipeline steps and allows easy scheduling.

[16] https://grpc.io/

- Batch execution: Airflow allows the scheduling of batch actions. Before, users had to white their own scripts to do it.
- Monitoring: the new integration allows users to easily detect failures in batch pipeline steps through the web interface and to configure e-mail alerts and other logical structures that improves monitoring.
- CI/CD support: the changes allow the developer to setup an easy way to schedule complex CI/CD pipelines, maintaining independent tasks to be executed in sequence, as long as the structure of a DAG is respected.

3.2.4 CLI Improvements

From the old architecture to the new one, the component that showed more visible improvements was Marvin-AI's CLI (Command-Line Interface). Commands to generate templates were vastly used to improve the deployment versatility, generating Kubernetes configuration files, or enabling the Apache Airflow integration, that relies mostly on CLI.

With this said, we can evaluate that the CLI improvements have benefited the following quality attributes:

- Usability: this quality attribute improved in general because the CLI has more tested code being generated now, thus the developer has to write less code to adapt the solutions to specific infrastructure configurations.
- Robustness: as the infrastructure code is generated by the CLI, the errors that may occur in a manual configuration process are no longer a concern. The files can be audited and versioned, occasional problems can be easily solved by changes on configuration files, instead of performing manual and undocumented changes on a large number of servers.

3.2.5 Benchmark suite

This section is a summarized version of a paper published at the 2020 IEEE International Conference on Machine Learning and Applications (IEEE ICMLA 2020) (SILVA et al., 2020).

Monitoring is an important quality attribute for machine learning, not only for aspects related to algorithm performance, but also resource consumption and execution aspects, such as task completion time, CPU and GPU usage, memory usage, disk input/output and network traffic (SILVA et al., 2020). There are already many tools available for monitoring these aspects

directly in the operating system, therefore there is no need to develop an alternative tool or library. But because Marvin-AI is supposed to help less experienced users, there should be easier ways to do it, and this was the focus of this architectural change. The approach is based on the following elements:

- *Docker* containers are used to run the application. As described in Section 3.2.1, containers facilitate different MLOps tasks, such as deployment and version control. It also encapsulates an application into a well-defined environment, making it easier to monitor resource usage from an external point of view. There are many tools for working with container monitoring, but in this work *docker stats* was adopted, as it is one of the easier ways to gather the necessary resource information from a container;
- A simple API was developed to allow the developer to easily mark points-of-interest (POI) inside Marvin-AI tasks. These POI will appear together with the final data, so that it becomes easy to establish a relation between the observed resources and the actual code. This allows to easily separate resource usage for each task and monitor task progress through specific points in time;
- *Matplotlib* is used to display the resource usage data graphically. It is a Python-based visualization tool that can display 2D graphics;
- Tools such as Apache *JMeter* can be used to conduct stress tests with the online tasks. *JMeter* can be configured to flood a server with continuous requests from multiple simultaneous threads, simulating high-demand scenarios. By combining a particular POI from Marvin-AI with the beginning and end of *JMeter* tests, it is possible to know exactly when these requests start and finish, making it easier to see the server's response to the simulated period of high demand.

In order to use the approach, the developer must configure the necessary tools and include API calls in specific parts of the code. Configuring the necessary tools is as simple as installing them and correctly establishing the code dependencies with a few commands. The API usage is illustrated in Listing 3.4.

Listing 3.4: Batch actions annotated with different points of interest (POI)

```
1 poi_marker = POIMarker('times_batch')
2 time.sleep(10)
3 poi_marker.add_poi('a')
4 run_preprocess()
```

```
5 poi_marker.add_poi('b')
6 time.sleep(5)
7 poi_marker.add_poi('c')
8 run_training()
9 poi_marker.add_poi('d')
```

First (line 1), a `poi_marker` variable is initialized to serve as a marker for adding points of interest (POI). Next, four POIs are added. POIs "a" (line 3) and "b" (line 5) mark the beginning and end of the preprocessing step. POIs "c" (line 7) and "d" (line 9) mark the beginning and end of the training step. Among these lines, a forced delay is introduced (lines 2 and 6) to guarantee that the monitoring API is capturing the resource utilization values for each task separately, as there is normally an interval of time for data collection. It also gives some time for the garbage collection to run, if it is the case. These forced delays are optional, but are recommended to make the visualization clearer, specially when the tasks run in a short period of time.

To monitor the execution, the developer must use the `benchmark` command in Marvin-AI CLI, that provides batch results in form of matplotlib's graphs. To use the benchmark suite in a online manner, the developer must use the command `http-server` at Marvin-AI CLI with the flag `--benchmark`. This will initiate a background process monitoring the execution of `engine-executor`'s API.

Now we show an example of the approach being used to monitor how an ML solution consumes the resources of a computer. The following examples are from a solution based on algorithm SVC (C-Support Vector Classification) for a dataset from kaggle[17]. Figure 3.2 shows an example for the resource usage data throughout time for the code of Listing 3.4.

It is possible to see that the execution took a little over 90 seconds, including the preprocessing and training tasks, and the forced delays. The four POIs ("a", "b", "c" and "d") are clearly visible, as well as the forced delays (before "a" and between "b" and "c"). This graphic shows that both tasks (preprocessing and training) are using a single core, as only 100% CPU usage is being registered (top left). Memory usage (top right) is more intense during preprocessing (8% peak) and smaller, but constant, during training (6%). It is also visible that disk I/O happens only during preprocessing (bottom left/right, between "a" and "c"), as the amount of data read/write does not increase during training (bottom left/right, between "c" and "d"). As expected, there is more disk input than output in these tasks, as shown by the vertical axis. It is also possible to see that there was some disk output during a forced delay (between "b" and "c" in the bottom right), probably due to buffered disk operation.

[17] `www.kaggle.com/c/santander-customer-transaction-prediction`

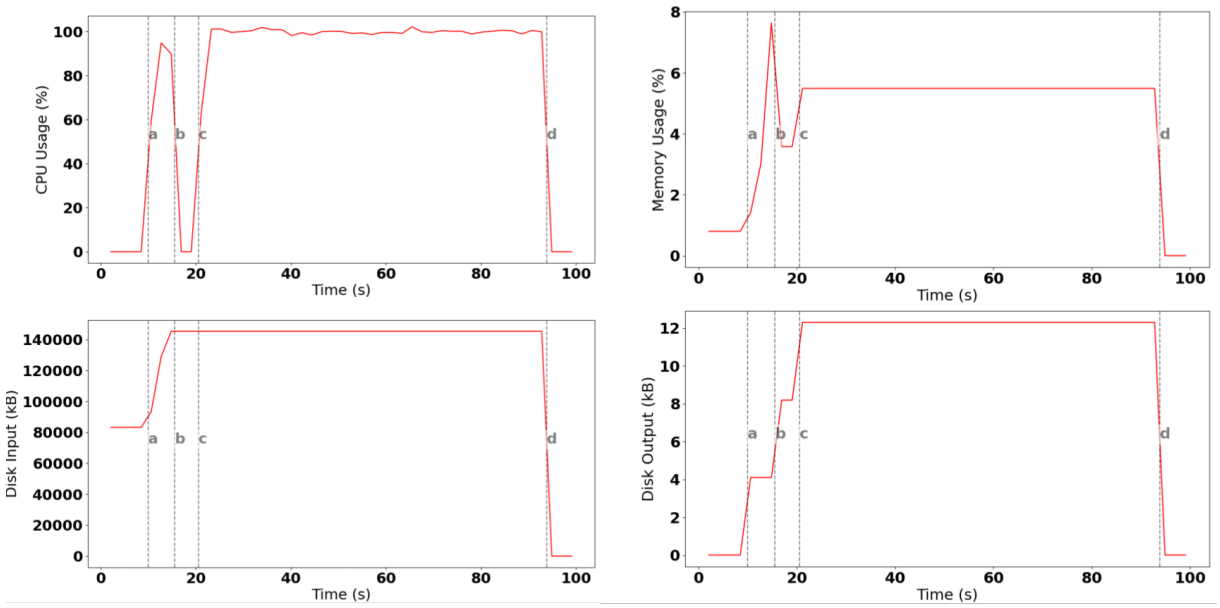

Figure 3.2: Resource usage (CPU, RAM and disk I/O) for the code in Listing 3.4.

The example of Listing 3.4 consists of batch actions. These are executed once every now and then, for preprocessing or training, for example. To monitor online tasks, it is necessary to consider a different setup.

Normally, an online task is part of another application, and must respond to requests through a REST or gRPC endpoint. Therefore, there is a server where these endpoints are being served, and a client, where these endpoints are being consumed. Also, online tasks should normally be quick to respond and support a certain amount of simultaneous requests without delays or failures. The exact amount of simultaneous requests depends on each business scenario. For example, a small company's ML solution being used only by upper management will probably have only a few simultaneous requests in the most busy moments, while a large company's ML solution being used by customers during holiday shopping may have peaks of thousands or tens of thousands requests arriving simultaneously.

Marvin-AI can also be used to monitor these tasks in both sides and considering these different scenarios. In conjunction with a stress test tool, such as Apache *JMeter*, Marvin-AI can help to visualize resource consumption in these conditions. First, the endpoint server must be configured to mark POIs using the API as shown before. At least two special endpoints are needed, one for the beginning and one for the end of the stress tests. In each endpoint, nothing is done except for adding a POI marker for these events.

Next the developer must configure the stress test tool (*JMeter*, for example) to simulate the desired scenario. For example, it may be configured to start 100 threads simultaneously sending

100 requests in sequence. It should also be configured to send a single request for each special endpoint mentioned before, before and after the tests. This allows the resource utilization to be better visualized. Ideally, this tool should be executed in a different machine than the server. It would be even better if many machines are used as clients, to avoid bottlenecks in the client and simulate more clients.

Figure 3.3 shows the results of a stress test with a client computer simulating 50 threads, each one sending 1000 requests in sequence to a server computer. This test took around 150 seconds to complete. The left side shows CPU usage in the server during this time, which operates mostly between 200 and 250% to respond to the requests. It is possible to see the POIs "stress_init" and "stress_end" delimitating the exact moments when the test begins and ends. The right side of Figure 3.3 shows the response times being observed in the client during this time. In this case, it is possible to see that most requests are being returned in a very short time, practically indistinguishable from 0 in the graph. *JMeter* also reports the number of errors, which in this case was 0%. These results are an indication that this particular server is being able to respond to this amount of requests properly, and should behave well in production under these conditions.

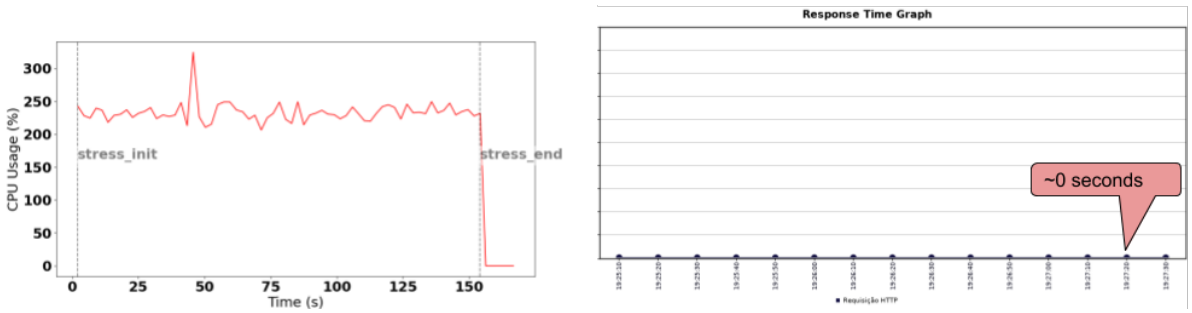

Figure 3.3: Example CPU and response time for a stress test of online tasks simulating 50 simultaneous requests.

In contrast, Figure 3.4 shows the results of a stress test simulating 1000 threads, each one sending 1000 requests in sequence. This test took almost 4000 seconds to complete. The left side shows CPU usage in the server during this time, which operates mostly between 100 and 250% to respond to the requests, but has sometimes reduced to 0%, which indicates that the server stopped working in several occasions. The right side of Figure 3.4 shows the response times being observed in the client during this time. In this case, it is possible to see that there are some requests that are taking too long to return, with some peaks reporting more than 10 seconds to respond and others taking more than 40 seconds. Also, there are too many requests that are close to the first horizontal line of the graph (the red line), which marks a 6-second response, normally unacceptable for a good quality service. *JMeter* also reports the number

of errors (requests without a response), which in this case was 5,36%. These results are an indication that this server is not being able to respond to this amount of requests properly.

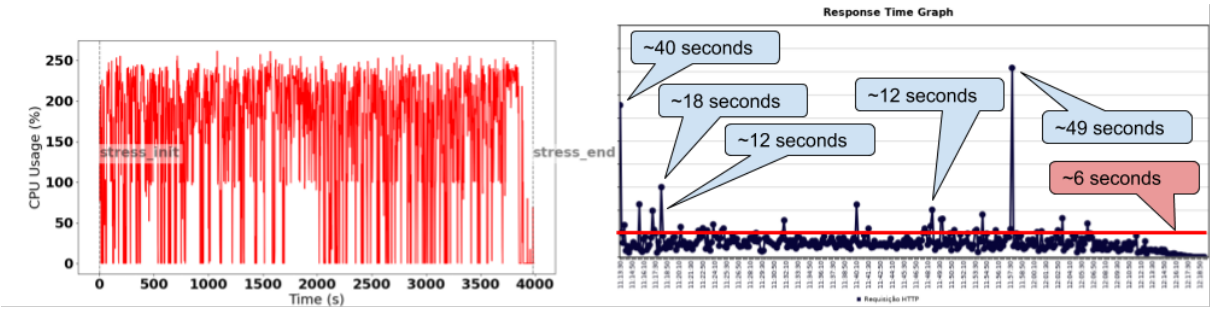

Figure 3.4: Example CPU and response time for a stress test of online tasks.

As another example of how this information could be used, Figure 3.5 shows the disk input and RAM usage for a solution using a label encoder[18] for the Microsoft malware dataset[19]. This execution was interrupted during preprocessing (POI "b" does not appear in the graphs). Disk input was high, but this is expected because it is a large dataset. The problem was the lack of memory, as the algorithm used constantly more memory over time, until it reaches close to 100% and the process is interrupted. In this case, either more memory is necessary or a different algorithm must be implemented for the preprocessing task.

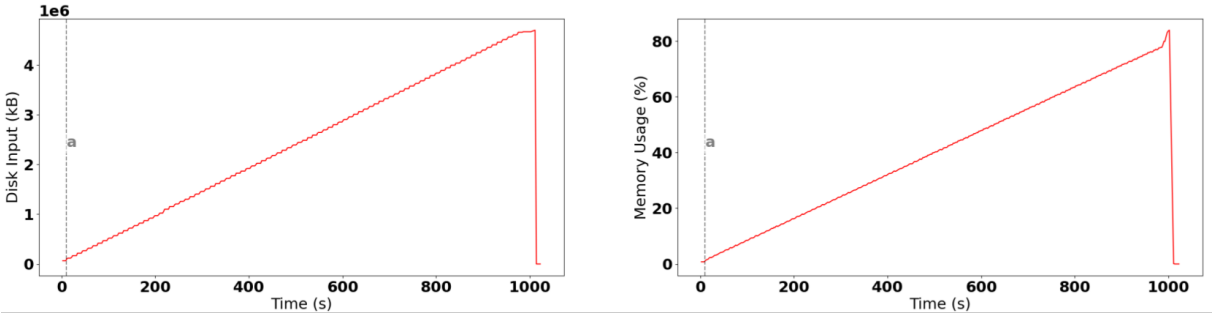

Figure 3.5: Disk input (left) and RAM (right) usage for the solution to the Microsoft malware dataset. The values are cumulative over time.

The integration of this benchmark suite has provided better support for the following quality attributes:

- Usability: monitoring these resources in Marvin-AI is considerably easier than using the tools by themselves, as all that is needed is a simple configuration and some CLI commands.

[18] `scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html`
[19] `www.kaggle.com/c/microsoft-malware-prediction/overview`

- Robustness: with Marvin-AI's new features it is easier to correctly provision the necessary resources so that the ML solution does not run out of CPU or memory in normal and critical conditions.
- Prototype to deployment: putting a solution in production requires choosing the right hardware, and this new feature allows this task to be conducted with more confidence on what is needed.
- Monitoring: this is the main benefit of this new Marvin-AI feature, which allows different resources to be monitored.

3.3 The architectural evolution process - a simplified version of ATAM

The list of architectural changes described in the previous section was not conceived in a single meeting or over a short period of time. Instead, it was incrementally constructed after many discussions and following a systematic process that involved different stakeholders. This section describes this process and details how all the changes were designed.

When the last stable version of Apache Marvin-AI platform was released, still in the old architecture, there were already some discussions about architectural changes. An architectural sketch (Figure 3.6) was defined in an ad-hoc way. It focused on introducing Docker containers into the engines. The sketch showed how the communication would work between the CLI and the Marvin-AI container, the shared volumes and the usage of Docker SDK[20] to maintain the containers. Other possibilities for interacting with the engines, such as REPL[21], were being envisioned too.

While there was an overall consensus about how Docker containers would bring benefits such as a greater language and technology independence and facilitated deployment, the developers were not fully aware of all details and implications of this new architecture. For example, how would other Marvin-AI's important features be affected, such as its mission of being easy to use by data scientists without deployment and operation experience? What other unforeseen benefits would be achieved by this new version? For this reason, before development began, a proposition was made to the Apache Marvin-AI community.

[20] https://docker-py.readthedocs.io/en/stable/
[21] `https://replit.com/`

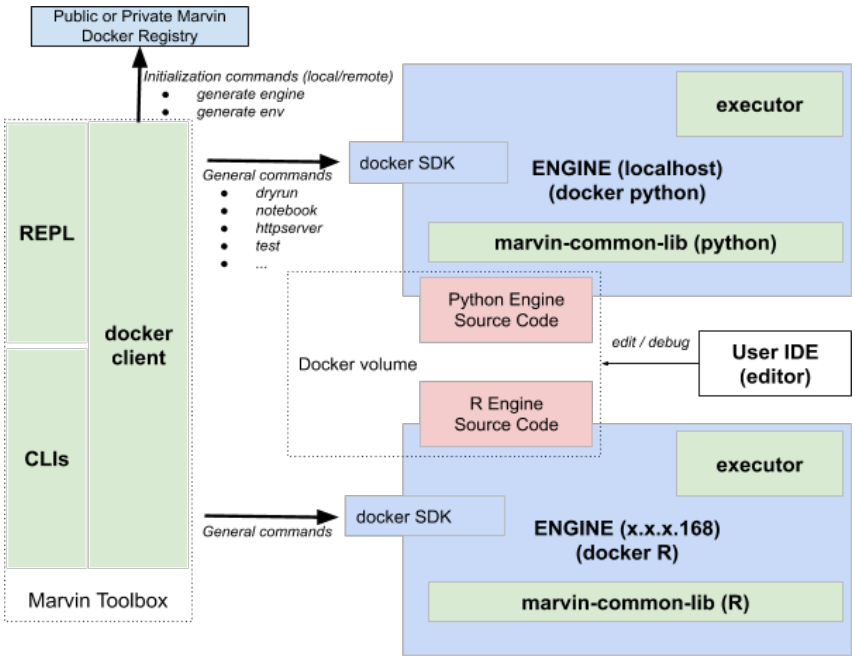

Figure 3.6: Marvin new architecture

The proposition was to use the Architecture Tradeoff Analysis Method (ATAM) (KAZMAN; KLEIN; CLEMENTS, 2000), a well-known method, to evaluate the proposed architectural changes before implementation began. It was expected that a more systematic process could help the developers to better identify the impact of the architectural changes over Marvin-AI's main mission and obtain more details of what was needed to implement the changes as effectively as possible. We also wanted to avoid architectural degradation caused by ad-hoc changes (BEHNAMGHADER et al., 2017; ALENEZI; KHELLAH, 2015).

ATAM is a method to evaluate software architectures (either implemented or idealized) by assessing some quality attributes. The method exposes the risks and tradeoffs that potentially limit the organization to achieve their business goals. In order to do that, it depends on the participation of all the important stakeholders in the process (KAZMAN; KLEIN; CLEMENTS, 2000).

The process starts with a presentation of the method and the software architecture to the stakeholders. The business drivers, the requirements from the corporation that will serve as a baseline for the architecture quality attributes, are also elicited with the participants (KAZMAN; KLEIN; CLEMENTS, 2000). Therefore, to begin, we needed one thing: to get all stakeholders together in a room and start the process. But because Apache Marvin-AI is an open source project, this was not simple.

As any other software incubated by the Apache Software Foundation (ASF), the process for general development discussions regarding Apache Marvin-AI is primarily driven by the

developers' mailing list[22] in conjunction with other tools, such as an issue tracker[23]. In general, to make a decision, a community member must start a new thread on the mailing list. Other members must present arguments and discuss the subject, following the terms stated by the code of conduct for the open source project and, at the end, conduct a poll if necessary.

A poll is formulated when there seems to be a consensus around a binary choice. Each member of the community that is participating in the discussion sends the message +1 to agree and -1 to disagree with the original proposal, being able to present a justification during the act. Everyone has the same vote importance in the development mailing list, but every project has a Project Management Committee (PMC) that is composed by leading members of the project. The PMC has its own mailing list for voting the more important issues, such as new releases and including other members in the committee.

As it can be seen, the entire process is asynchronous, with each member contributing to the project during free time. Also, it is difficult to identify all stakeholders, as the community does not enforce strict roles within the project. Although the PMC seems to be a strong candidate for these discussions, ideas could come from any member, as enforced by the egalitarian voting discussed before.

For these reasons, we introduced the first two simplifications to ATAM:

- First, because it is difficult to identify all stakeholders with potential to contribute, all members of the developers' list could participate in the process. Marvin-AI's community is not very big, so we did not expect problems such as long and unfocused discussions; and
- Because there are contributors from different countries, it would be challenging to arrange a synchronous meeting where everyone would be available. Therefore, all communications should be asynchronous, whenever possible.

With these two simplifications we started the process.

3.3.1 Process start and mission

In this stage, in our simplified version of ATAM, we elaborated a series of e-mails, sent to the developers' list between 07/07/2020 and 08/15/2020, to start the process: first we explained

[22] dev@marvin.apache.org
[23] `https://issues.apache.org/jira/projects/MARVIN/issues`

how the evaluation would occur, how long it would take, and what was expected from each participant. This e-mail was only informative and required no action from the stakeholders.

Next we had to define the organization's business drivers (KAZMAN; KLEIN; CLEMENTS, 2000). But because Marvin-AI is not supported by any particular organization, it does not have clear business requirements. Instead, we started a discussion to establish what is Apache Marvin-AI's mission. Although this was more or less clear in the website, it was important to write it down so that everyone would agree on it. After some discussion, the agreed statement came to be the following: The Apache Marvin-AI platform aims to offer a practical and standardized solution to help its users to perform data exploration, model development and application lifecycle management for artificial intelligence tasks, aiming to offer: scalability, language agnosticism and a standardized pipeline.

3.3.2 Old and new architecture - first improvement on the original sketch

Next, to make sure that everyone was fully aware of how the architecture was before, and how it would change, we started a second series of e-mails, from 08/15/2020 to 08/26/2020. We felt that Marvin-AI's developers already had a good understanding about the old architecture, but we developed some diagrams to formalize it anyway. We used UML class and component diagrams and shared them with the participants. There were not many responses in this regard, thus confirming that the developers already had a good understanding.

For the new architecture, we already had the sketch from Figure 3.6, which described some of the proposed changes. We shared it again with the participants and asked them if there was any doubts of what was being proposed. Again, there were not many responses, and we considered that everyone had a good understanding regarding this proposal.

At this point, the discussion raised an important issue that was not being considered before. Because the new architecture would adopt Docker containers, it would be possible to quickly start and stop instances of the engines (which contain the ML prototypes/solutions being developed), but also of the Marvin-AI tooling itself. This would allow deploying a Marvin-AI development environment in a container, facilitating configuration and even allowing remote development. However, for this to work, there had to be some way to keep the container instance running. Differently from the engines and the executor, which have GRPC servers and an HTTP server running, the Marvin-AI tools are simple CLI commands that respond to user interaction on a terminal. This is not enough to maintain the container running. The developers then came to the conclusion that the new architecture should use a Marvin-AI active component running in background (a daemon process) instead of `marvin-common-lib`. This would

allow the user to interact with Marvin-AI tooling and perform the same tasks as before, but in a container. The new containerization approach would also make it easier to monitor hardware resources.

These discussions were all made by e-mail, and resulted in the changes described in Sections 3.2.1 and 3.2.5. At this point, we were ready to move to the next stages of ATAM.

3.3.3 Quality attributes and scenarios - a synchronous meeting that did not go as planned

According to ATAM, the next step was to generate a quality attribute utility tree. This step has the purpose of compiling system utility, such as availability or performance, in terms of detailed quality attributes, and organizing them in the form of a tree. In the next stages, the tree is updated with scenarios that are used to assess the utility represented in the structure and define priorities for each node. Each scenario would represent a typical use case, with enough details to allow assessing how the architecture would respond. In the original ATAM, these steps are also supposed to be performed in a synchronous meeting, and had to be adapted here.

We were afraid that a list of desired quality attributes would not spontaneously arise from scratch. Therefore, we decided to start this process with a literature review, which led to the identification of the main quality attributes described in Section 2.2. We then shared this initial list via e-mail with the developers to obtain feedback on what was considered important to Marvin-AI.

There was a lack of response from the developers. Some expressed that they were having a hard time understanding these complex subjects in e-mail messages, as this short text style was not enough to fully grasp all the necessary details. They suggested that it would be better to do a video chat for this and the subsequent steps. Although we were attempting to keep all communications asynchronous, we happily agreed, because it was a suggestion made by the developers themselves and an opportunity to improve the discussions.

We then invited all interested developers for an online meeting. Following the suggested ATAM steps for this stage, this online meeting had the purpose of reviewing the quality attributes list and coming up with a set of scenarios for future evaluation. We prepared a set of presentation slides to conduct the meeting.

Six developers attended the meeting. We started by presenting the list of quality attributes. Although there was not much discussion regarding them, the developers agreed on their relevance. But after that, the meeting did not go as planned. Instead of discussing and prioritizing

scenarios, the developers moved back to a previous step: formalizing the new architecture. Although we thought to have already settled upon the new architecture, the developers decided to use this opportunity to plan more changes for the future.

We did not want to interrupt the creativity of the developers, therefore we followed the discussions. Some of the developers cited TFX and Apache Beam as open source tools that could be incorporated in Apache Marvin-AI to fulfill some of the requirements that Marvin-AI had not yet implemented. TFX has native metadata gathering, rollback executions, infrastructure checking and standardized environment (TFX, 2021). Apache Beam defines a domain specific language (DSL) to deal with data preparation at scale (APACHE, 2021a). The decision was that the next steps had to take into account the integration of other tools, whenever possible, to make them easier to use and thus honoring Marvin-AI's mission of making MLOps more accessible.

After this discussion, three new changes were incorporated in the new architecture: (1) the implementation of TFX wrappers, to provide a ease-to-use solution to certain kinds of tasks; (2) the usage of Apache Airflow to schedule a pipeline with DASFE actions; and (3) improvements on CLI. Together with Docker containerization, which was already planned before this whole process (Figure 3.6), and the first improvements (Section 3.3.2), we ended up with the consolidated set of changes described in Section 3.2 for the new architecture.

At this point, from the two goals for this first meeting (consolidating the list of quality attributes and identifying scenarios), only the first was being achieved. The second goal was not being achieved, and worse, we moved back to a previous step: formalizing the new architecture.

But before the meeting ended, we still wanted to plan the next steps and define scenarios for future evaluation, following ATAM. The developers then suggested that we simplified this stage to use a document format known as a "Request For Comments" (RFC). The RFC format has been used in the Internet Engineering Task Force (IETF) organization for the construction of the patterns for Internet protocols such as audio codecs[24] and also in open source projects such as Tensorflow (TENSORFLOW, 2021), to discuss the implementation of critical features. This would be performed asynchronously. Because the RFC format allows the inclusion of more details than in e-mails, we thought this would avoid the understanding problems mentioned before.

[24]https://www.rfc-editor.org/rfc/rfc6716.txt

3.3.4 First version of the scenarios - the RFC approach

In the ATAM method, there is a stage to brainstorm and prioritize scenarios (KAZMAN; KLEIN; CLEMENTS, 2000). The scenarios are composed by tree main parts: (i) *stimulus* is the interaction with the system made by another system, a user or itself; (ii) *environment* is what goes on with the system when the *stimulus* is made, including events such as status changes and interactions between architecture components (KAZMAN; KLEIN; CLEMENTS, 2000); and (iii) *response* expected by the architecture components, given a *stimulus* and contextualized in a *environment*. Computational resources or other related metrics are also used to evaluate the responses about certain quality attributes. The ATAM scenario framework also includes some other features to made a risk and trade-off analysis of each architecture decisions that the scenario interacts with. The decisions can be classified as risk/non-risk/sensitivity points/trade-off points (KAZMAN; KLEIN; CLEMENTS, 2000).

As discussed before, these scenarios were supposed to be defined during our last online meeting, but this was moved to an asynchronous moment, via an RFC document. At this point, we were already involved in many discussions within the Apache Marvin-AI community. Besides the emails in the developers' list, there were other, unrelated meetings, that involved some of the project's stakeholders. Therefore, we were able to define different scenarios based on all the discussions that took place until this moment. In the end, we still managed to collect the viewpoints of different stakeholders, but again we deviated from ATAM, because instead of using a single focused meeting to identify and prioritize scenarios, we compiled this information from previous meetings.

Each scenario purposely defines critical contextual information, such as the user's knowledge level, infrastructure configuration and use of third-party tools. This is important because different quality attributes may be evaluated differently depending on the context. For example, the most important quality attribute to a beginner can be standardization, because it eases the learning curve, but an advanced user can get more value from an extensible solution.

This resulted in three scenarios being included in the RFC, summarized as follows:

- The **first scenario** describes the basic usage of Marvin-AI's TFX integration, when a data scientist writes his code to develop a model using the TFX framework inside Marvin-AI, performs a basic integration test and finish with a model ready to deploy. As Marvin-AI is designed in his new architecture to provide a prototype environment close to the ideal performance, we considered that the interactive development inside notebooks is a non-risk. A trade-off point between the this type of development and the pipeline orchestration

was highlighted as more complex pipelines must be robustly orchestrated. Interpretability is lost at the expense of performance and continuous training.

- The context of the **second scenario** is the remote development with Marvin-AI. The developer can operate Marvin-AI through its daemon process running inside a container. The DASFE design pattern provides programming support for TFX framework, Marvin-AI standalone solution or a mixed version of both. The usage of Marvin's actions allows the developer to insert code outside the TFX components in the pipeline. This is viewed as a non-risk, because it allows a better integration of TFX with external solutions. A trade-off point was detected in allowing remote development outside a fully managed platform, as it requires extra configuration skills generally not possessed by data scientists (FINZER, 2013). On the other hand, this type of tool gives the user more autonomy in relation to the computational cost of the algorithms used.
- The **third scenario** describes the Airflow DAG generation, highlighting the fact that this generation allows the developer to easily extend the DAG, thus being considered a non-risk. A sensitivity point found was the insecurity about scheduling the pipeline components without a precise prevision of how much time it takes to run them.

We formalized the new architecture, the quality attributes and the scenarios into an RFC document. We followed the Tensorflow RFC model, but adapted it to work with ATAM scenarios. Our RFC was composed by: (i) motivations and context to the implementation; (ii) quality atributes regarding the scenario; (iii) all the architecture changes and interactions intended; (iv) the scenarios descriptions in the ATAM model.

The RFC was sent to the development mailing list. We asked a feedback about the proposed scenarios, new possible scenarios, trade-off points, risks and non-risks, sensitivity points and new related quality attributes. Although we asked several times, we did not had any answer after a couple of weeks.

3.3.5 Second attempt with RFC - simpler scenarios and a video

At this point in time, the evaluation was not complete yet, but the developers had already implemented most of the proposed changes as a separate branch in Marvin-AI's repository. Even though this is not recommended by ATAM, the dynamic, fast-paced nature of this project led to this result. Nevertheless, the changes were still on a separated branch, unapproved, and we proceeded with the evaluation of the new architecture, even though it was not going exactly as planned by ATAM.

Because there was no feedback from the first RFC attempt, we tried to make things simpler. We removed the details (priorities, risk, non-risk, sensitivity points and trade-off points) from each scenario and refined them a little more. We ended up with five different simplified scenarios, all of them with the same priority:

- A data scientist with no knowledge of how to deploy an ML solution wants to use Marvin to experiment, test, find a good model. At the end, the data scientist wants to put the solution into production;
- A developer or data scientist wants to use Apache Marvin-AI, but does not have a good machine available locally or does not want to use her own machine to develop;
- A team with experience in deployment, who already knows how to put machine learning solutions in production, wants to use Marvin to facilitate the process of deployment and continuous delivery;
- The team already has some components ready and wants to replace some components of the Marvin pipeline, or modify its structure/pipeline to integrate Marvin and other tools;
- The team uses different tools, languages and technologies for different stages of the pipeline, but wants to avoid unwanted complexity using Marvin-AI.

To help developers to better understand the proposed changes and better associate each scenario with the quality attributes, we produced a video presentation that describes how the new version of Marvin-AI would be used. The fact that the new version was already mostly implemented helped to produce a video that delivered a full "hands-on" experience, thus facilitating the evaluation.

We then sent the RFC again, the video and a questionnaire that asked the developers to evaluate each of these five new scenarios in terms of the identified quality attributes. Each question targeted a specific quality attribute, to diagnose if the new architecture is better or worse in regards to that attribute. Each question had two parts: the first part was a multiple-choice question following a Likert scale. The second part asked the developer to justify her answer.

This simplified approach was better received, and we had five responses from this evaluation. This is not a large number, but considering that the Marvin-AI community is small and the respondents were all Marvin-AI experts, we considered this as a success. The results of this evaluation, together with other forms of evaluation, are presented later in this essay.

Chapter 4

EVALUATION

The developers and PMC members that were present at this meeting agreed that the changes that resulted from this process were very positive and moved Marvin-AI to the right direction, which is to reuse what is already available in other tools without sacrificing Marvin-AI's mission. But further steps should be taken to modernize Marvin-AI's architecture even more. Some community members had the idea of elaborating a new RFC describing a solution that would add even more flexibility to the pipeline, making DASFE pattern not mandatory, make it possible to run Marvin-AI engines and tooling on top of Kubernetes[1] and to allow more modularized container components.

Marvin-AI's new architecture was evaluated from 4 different perspectives. The first one was the feedback given by Marvin-AI developers in the ATAM evaluation, as described in the end of the previous chapter. It analyzes how the new architecture has improved considering the quality attributes explored in Section 2.2. The second evaluation was a code analysis, to gather quantitative evidence of quality attributes related to code quality: (i) usability; (ii) prototype to deployment and; (iii) component reuse. The third one uses the taxonomy proposed by Lwakatare et al. (2019) to evaluate the maturity that Marvin-AI's new architecture can bring to its use cases, when compared with the previous one, also bringing more evidence to confront the developers' opinions. And the fourth one evaluates Marvin-AI's new resource monitoring features.

4.1 ATAM Evaluation

The simplified ATAM process produced the following artifacts:

[1] https://kubernetes.io

- The initial architectural sketch;
- A set of quality attributes validated and prioritized by the Apache Marvin-AI community;
- New architecture and third-party tools integration;
- The produced RFC, video and questionnaire that contain Marvin-AI's mission, architectural descriptions, quality attributes and scenarios;
- Mailing list logs containing discussions regarding the initial architectural sketch and further proposed changes, the mission definition, the discussions following the simplified ATAM process being followed and other discussions; and
- A paper and this dissertation that summarize the entire research, present the main results and contributions to the academic community. By the time this dissertation was being written, the paper had not been published yet.

Following the final ATAM step, which is to package the results (KAZMAN; KLEIN; CLEMENTS, 2000), all these artifacts are available in the project's official Github repository[2], and official mailing lists/issue tracking system.

At the end of the process, another online meeting was scheduled with some stakeholders to discuss Apache Marvin-AI's future. This time, we chose this format instead of e-mail discussions, as we thought it would be more productive, and although not many developers were present, some fruitful discussions were made.

As described in Section 3.3, we sent an RFC document and a video describing the new architecture. The document contained five scenarios to be evaluated by the developers. Each scenario had some questions to be answered in terms of how the new architecture influences the quality attributes.

Each question had different answers, but they all followed the same pattern: how much better or worse is the new architecture, in regards to a particular quality attribute? We normalized the results to: (i) much better; (ii) slightly better; (iii) no difference; (iv) worse and; (v) much worse.

Not every quality attribute was evaluated for all scenarios, since in some cases it makes no sense. For example, in the first scenario, component reuse is not an issue, therefore it was not evaluated. But by combining the five scenarios, we managed to evaluate all quality attributes, as shows Table 4.1.

[2]https://github.com/apache/incubator-marvin

Quality attribute	Scenarios				
	1	2	3	4	5
Usability		X			
CI/CD Support	X				
Robustness	X				
High availability	X				
Standardization	X		X		
Component reuse					X
High performance	X				
Prototype to deployment		X		X	
Monitoring			X		
Extensibility				X	X
Batch execution			X		

Table 4.1: Quality Attributes evaluated in each scenario

Quality attribute	Evaluation	#
CI/CD support	Much better	4
	Slightly better	1
Standardization	Much better	3
	Slightly better	1
	No difference	1
Robustness	Much better	5
High performance	Much better	4
	Slightly better	1
High availability	Much better	3
	Slightly better	2

Table 4.2: Scenario 1 results

In the end, 5 developers answered these questions. According to ATAM (KAZMAN; KLEIN; CLEMENTS, 2000), this number is adequate, as they are all Marvin-AI experts. Next we present their responses.

The first scenario was: "A data scientist with no knowledge of how to deploy an ML solution wants to use Marvin to experiment, test, find a good model. At the end, he wants to put the solution into production". Table 4.2 summarizes the results, which evaluated five quality attributes: CI/CD support, standardization, robustness, high performance and high availability.

All developers think that Marvin-AI's new architecture has a better support to CI/CD workflows, disagreeing only in terms of how much better it is. The primary justifications was that it now has better integration with large-scale scheduling systems and TFX.

The majority of the developers also thinks that Apache Marvin-AI is more standardized now, due to the generation of standardized DAGs and deployment. One developer considered

Quality attribute	Evaluation	#
Usability	Much better	4
	Slightly better	1
Prototype to deployment	Much better	2
	Slightly better	3

Table 4.3: Scenario 2 results

that the standardization remains the same, because with the new features that Marvin-AI has now, there is space for more abstractions and support for even easier commands.

Regarding robustness, all developers agreed that the new architecture is much more robust now, given the integration with Kubernetes and TFX.

The last two questions of this scenario are related with the high performance and high availability quality attributes. All developers agreed that Marvin-AI is more capable to deliver higher performance in this new architecture, although one of them considered it only a minor improvement. The justifications mention the better integration with Kubernetes and Tensorflow, both high-end industry solutions prepared for high performance.

Regarding high availability, three developers considered that the new features greatly improved this quality attribute, while two of them defended that it was only a slight improvement. The justification given was that it is simpler to deploy Marvin-AI on Kubernetes, since several steps of the process were removed with the containerization of the engine and deployment files generation.

Scenario 2 was: "A developer or data scientist wants to use Apache Marvin-AI, but doesn't have a good machine available locally or doesn't want to use his own machine to develop". As shows Table 4.3, two quality attributes were evaluated: usability and prototype to deployment.

Regarding the usability quality attribute, all developers agreed that the new version of the architecture has improved. The main justification is that the CLI can abstract the process of a remote environment, which is a time costly task to do manually.

Another quality attribute related to remote development is the easy migration from a prototype solution to deployment. All the developers agreed that the new features closes the gap between the two environments, although the majority of them defended that it is only a slight improvement. The major justifications are that, while the solutions do work, to fully address the problem, a better integration with conventional CI/CD tools is needed.

Scenario 3 focuses on the workflow of a team with experience in deployment, who already knows how to put machine learning solutions in production, and wants to use Marvin-AI only to

Quality attribute	Evaluation	#
Standardization	Much better	3
	Slightly better	1
	No difference	1
Monitoring	Much better	4
	No difference	1
Batch execution	Much better	2
	Slightly better	3

Table 4.4: Scenario 3 results

Quality attribute	Evaluation	#
Extensibility/Prototype to development	Much better	1
	Slightly easier	4

Table 4.5: Scenario 4 results

facilitate the process of deployment and continuous delivery. Tables 4.4 summarizes the results, which evaluated three quality attributes: standardization, monitoring and batch execution.

The majority of the participants responded that Marvin-AI's new architecture is more standardized regarding the end-to-end ML workflow, except for one of them, who responded that both architectures have the same support for this attribute. The main justification for the improvement was that Marvin's new architecture has new features and a large scale deployment option in its tooling now.

The second quality attribute in this scenario is monitoring. The majority of the developers agreed that Marvin-AI's new architecture has improved significantly regarding this attribute because of the Airflow and TFX integration, adding more monitoring capabilities and process visualization features, like metadata gathering and a web visual interface. One of the developers was not comfortable enough with presenting an opinion here and chose "no difference".

The last quality attribute considered in scenario 3 is batch execution. All developers agreed that the new architecture is better at executing batch tasks and monitoring them. Apache Airflow integration was mentioned as the responsible for the improvement. Some of them argued that, although the changes are positive, it would be better to integrate with something more abstract than Airflow, citing Elyra[3] as an example.

Scenario 4 was: "The team already has some components ready and wants to replace some components of the Marvin pipeline, or modify its structure/pipeline to integrate Marvin and other tools". As shows Table 4.5, two quality attributes were evaluated: extensibility and prototype to deployment. They were evaluated together in this scenario.

[3]https://github.com/elyra-ai/elyra

Quality attribute	Evaluation	#
Extensibility	Much better	1
	Slightly better	4
Component reuse	Much better	2
	Slightly better	2
	No difference	1

Table 4.6: Scenario 5 results

Four developers considered that the new architecture had a slight improvement over the old one, and one said that this improvement was major. The main justification is that, while the features mean a lot for Tensorflow users, Marvin-AI still has some problems regarding dependency management, that were not addressed yet.

Scenario 5 was: "The team uses different tools, languages and technologies for different stages of the pipeline, but wants to avoid unwanted complexity using Marvin-AI". As presented in Table 4.6, in this scenario two quality attributes were evaluated: extensibility and component reuse.

Regarding extensibility, all developers agreed that Marvin/AI has improved in the new architecture, differing only in terms of intensity. The major feedback points were the new Tensorflow integration and that the new architecture is more extensible than the other, although some of them pointed out that the lack of documentation did not allow Marvin-AI to achieve its full potential in this quality requirement.

For the component reuse quality requirement, the majority of developers considered that Marvin/AI has improved, but one of them didn't agree that the new architecture has any difference in this context. The major feedback points are that while the new features, such as DAG generator and other new CLI commands, facilitate the reuse of pipeline components, the structure seems the same.

Overall, based on the feedback, the evaluation confirmed that the new architecture have improved in terms of all quality attributes. It also elicited some possibilities for further improvement:

- There is space for more abstractions and support for even easier commands;
- It takes too much time to deploy Marvin-AI on Kubernetes;
- A better integration with conventional CI/CD tools is needed;

- It would be better to integrate Marvin-AI with something more abstract than Airflow, such as Elyra; and
- Marvin-AI still has some problems regarding dependency management and lack of documentation.

4.2 Static code analysis

We wanted to confront the results from the previous section with some objective measurements, to either confirm or contest the subjective opinions from the developers. Although these developers are the experts in Marvin-AI development, an objective confirmation would serve as additional evidence towards the effectiveness of our architectural evolution.

We decided to evaluate Marvin-AI's new architecture by building 2 solutions that are commonly used in industrial and academic environments and by performing static source code analysis. For this analysis, we focused only on the Marvin-AI/TFX integration, because this is the only new feature that would present differences in terms of source code.

The evaluation consisted in the following procedure: we developed the same ML solution in three versions: Marvin-AI's original architecture, Marvin-AI's new architecture (with TFX integration) and a standalone TFX environment.

Next we collected some static source code metrics:

Number of logical lines of code (LLOC). The number of logical lines of code. Every logical line of code contains only one statement. The more the LLOC count, the more effort it is necessary to create and maintain the source code.

Halstead Metrics. The Hastead metrics aim to identify measurable properties and identify relationships. These are the properties statically calculated in the source code: (i) the number of distinct operators (n_1); the number of distinct operands (n_2); the total number of operators (N_1); the total number of operands (N_2) (COLEMAN et al., 1994).

Using these properties, these relationships were explored:

- **Difficulty:** $D = \frac{n_1}{2} * \frac{N_2}{n_2}$ (larger values indicate more difficulty in developing the code)
- **Volume:** $V = Nlog_2 n$ (where $N = N_1 + N_2$ and $n = n_1 + n_2$)
- **Effort:** $E = D * V$ (larger values indicate that more effort is needed to write the code)

These metrics were collected with the help of `radon`[4], a freely available source code analysis tool for Python code and Jupyter notebooks. We only analyzed those pieces of code that are actually edited by the developer in the Jupyter notebooks. Generated code, such as the different files generated by Marvin-AI to organize the engines' components, was not accounted for, as it is not visible to the developer.

These are not definitive or very reliable metrics, but they can give some indication on the maintainability and simplicity of the code, specially when analyzed in conjunction with other metrics. In our case, we are using these metrics to confront the results from a different evaluation, therefore we expect them to either reinforce or contradict the subjective perception of Marvin-AI developers.

Regarding the quality attributes, the major objective is to reduce the trade-off between the **robustness** of a solution, as the qualitative analysis shows that TFX has the ability to build pipelines that has more quality regarding this attribute than the pure Apache Marvin-AI, and **usability**, which is one of Marvin-AI's strong aspects.

4.2.1 First solution - MNIST

Regarding the first solution to be used in this analysis, Li et al. (2014) describe an application of image classification that uses a customized framework for Convolutional Neural Networks (CNN). CNNs have been tested among a variety of datasets for image classification tasks, some are: MNIST, CIFAR-10 and NORB database (LI et al., 2014).

The MNIST dataset provides a relatively simple image classification task. It is composed by a set of handwritten digit images in conjunction with each image class (digits from 0 to 9) (Deng, 2012). Although the task may be simple, and can be solved by a variety of algorithms, this dataset provides a concept proof about a common industry problem, putting image classification models into production.

In this type of task, it is necessary to encode the image to transform it in a format that fits as model input. This has a high computational cost, as this operation has to be executed both in a batch model training scenario and at run time when serving the model predictions trough a REST API, which could potentially generate a massive overhead during model serving.

The Apache Marvin-AI community has made available an example[5] showing how to develop an image classification example using CNN to recognize handwritten digits trained by

[4] https://pypi.org/project/radon/

[5] https://github.com/apache/incubator-marvin/tree/develop/public-engines/mnist-keras-engine

MNIST. This was the **first version** used for the source code analysis. In this example, the implementation includes code for all of DASFE's batch actions: (i) Acquisitor and Cleaner, which has the code to use the Keras[6] MNIST load function and persist the dataset using a simple variable attribution; (ii) Training Preparator, which has the code to reshape the images in an adequate form to serve as input to the algorithm, and persist the final dataset; (iii) Trainer, which uses Keras to compose a CNN defining the layers, activation function, optimization method, among others, persisting the model in the end; and (iv) Metrics Evaluator, which extracts the metrics from the CNN training and persists them.

The Prediction Preparator online action, in this scenario, is responsible for acquiring the image from the server and making the transformations required for using it as model input. It is a common industry practice to use another service to handle image uploading, in compliance with the principles of a microservice architecture. Finally, the Predictor online action uses the input transformed by the Prediction Preparator action and returns the predicted image class (0 to 9 digits).

For the **second version** used for the source code analysis, we adapted the previous solution to use Marvin-AI's new integration with TFX. This introduced some particularities in the composition of the steps. TFX has a bigger variety of components to compose a production ML pipeline, easier to integrate in a CI/CD environment. In Marvin-AI's wrapper it is expected that a given DASFE action assumes a set of TFX components each.

The first TFX component used, on the Acquisitor and Cleaner DASFE action, is `ImportExampleGen`. It reads a file in the TFRecord[7] file that contains a transformed MNIST dataset, shuffles it and splits in training and test samples, as it is a good practice in machine learning.

The next set, defined in Training Preparator DASFE action, began with `StatisticsGen`, that uses only the output of `ImportExampleGen` as parameter to generate statistics for data visualization and example validation. These statistics are a parameter for the `SchemaGen` component, which generates a schema and can infer the shape's features and data types. Another such component is `ExampleValidator`. It uses the statistics and the schema to evaluate anomalies automatically on the dataset, using some heuristics. The last component defined in Training Preparator action is `Transform`. It uses the examples, schema and a function defined by the user called, by default, `preprocessing_fn`. The transformation applied in the MNIST dataset is a simple scaling, using the Tensorflow Transform library.

[6] https://keras.io/

[7] A custom file format defined by the Tensorflow libraries to support data persistence in a optimized JSON based structure.

The next to execute is the Trainer component, the only component in Trainer DASFE action. Besides the default format, TFX allows users to choose a variety of model formats, including: Tensorflow Lite, which aims at smartphones and IoT devices, and Tensorflow JS, which provides embedded use on Javascript interpreters. The Trainer component is standardized, as the training algorithm is written by the developer in a separate file. The component defines the examples, transformation model, schema and other training arguments as parameters (TFX, 2021).

As more than one Trainer component can be used to train models in different formats and objectives, components `Evaluator` and `Pusher` work in a similar way, as both can be present more than once in the pipeline, in the Metrics Evaluator DASFE action, with different configuration and parameters. The `Evaluator` component makes sure that, in a CD pipeline, the delivered model is always superior to the previous one. This task is called "model blessing" and must run on every model produced inside TFX before being deployed into the production environment. The last component of a typical TFX pipeline is the `Pusher`. It simply verifies if the given model is blessed and persists it in a user-defined file system location.

The **third version** of this solution is very similar to the second one in terms of code structure, the only difference is that the pipeline is orchestrated by TFX native tools. Some additional code and configurations are required.

Table 4.7 shows the metrics from the three versions of the MNIST solution. In terms of logical lines of code, the pure Marvin-AI version is the simpler, with around half as less code than the other versions. This is expected, as TFX introduces more components into the solution, thus requiring more code. In Marvin-AI + TFX, even more lines of code are needed, because each component of a Marvin engine must read the metadata from the previous component in order to follow DASFE, which is not required in pure TFX.

Solution	LLOC	Difficulty	Effort
Pure Marvin-AI*	65	1.1	10.37
Pure TFX	119	1.5	13.99
Marvin-AI + TFX	129	1.0	11.60

Table 4.7: MNIST dataset solutions metrics (* import and super class omitted)

Although the LLOC analysis seems to suggest otherwise, the difficulty and effort metrics show that the new Marvin-AI architecture with TFX is simpler to program and requires less effort than the pure TFX version. It is even less difficult (but requires more effort) to program than a pure Marvin-AI solution, which in theory is the simpler one. This is explained by the fact that Marvin-AI hides some of TFX's configuration code.

4.2.2 Second solution - MRPC

This model is generated by executing transfer learning (GOODFELLOW; BENGIO; COURVILLE, 2016) from a pre-trained BERT (DEVLIN et al., 2019) model in a semantic textual similarity analysis task using the Microsoft Research Paraphrase Corpus (MRPC) dataset. The **first version**, with pure Marvin-AI, uses the `fast.ai` framework[8] to do this task. The code is very high level and simple since the library contains all the tokenizers, training functions and metric generators.

With the **second version**, the solution only significantly differs from the previous one (MNIST) on the Transform and Trainer components, because here the developer must define new code. In this dataset, for containing expressions in natural language, the data has to be tokenized. In the training stage, the developer defines a module file containing the layers for the tokenized input, a code that downloads BERT's pre-trained network from TensorflowHub and adds it as a layer onto a new network and the output layer to classify if the two expressions that are part of the input have similarity between them.

The **third version** is identical to the second one, except for Marvin-AI's built-in libraries and the orchestration code that uses TFX's native libraries to configure the pipeline orchestration.

Table 4.8 shows the metrics for the three versions of the MRPC solution. The results are consistent with the ones from Table 4.7: again, the pure Marvin-AI solution is the simplest. The difficulty and effort metrics were calculated as 0 for pure Marvin-AI because the code has no operators nor operands, only attributions, which are not considered as operators by `radon`. The new Marvin-AI architecture, with the TFX wrapper, requires more lines of code, but is easier to program and takes less effort than a pure TFX solution.

Solution	LLOC	Difficulty	Effort
Pure Marvin-AI*	60	0	0
Pure TFX	130	2.9	115,39
Marvin-AI + TFX	133	2.38	104.78

Table 4.8: MRPC dataset solutions metrics (* import and super class omitted)

4.2.3 Discussion

In summary, in both datasets that had code evaluated, the Marvin-AI solutions, with and without TFX, are easier to program and take less effort, according to the metrics. These results indicate that the Marvin-AI integration with TFX managed to bring the benefits of TFX without

[8] https://www.fast.ai/

sacrificing Marvin-AI's mission of being easy to program. This result **confirms** the subjective perception of the developers in the previous evaluation, which stated that the new architecture is still easy to use, when compared to the old one, and it brings benefits in terms of robustness, performance, reuse and other quality attributes that came from the TFX integration.

This static code analysis can provide a quantitative basis to analyze if the trade-off between **robustness** and **usability** mentioned before has been reduced with the integration of Apache Marvin-AI and TFX. The metrics for both datasets (MNIST and MRPC) indicate that the pure Marvin-AI solutions take less effort and have less difficulty involved in the coding process. The Pure TFX solutions are in the other extreme, as the metrics indicate that they take more effort and have more difficulty involved in the development. In the middle are the Marvin-AI + TFX solutions, which presented better metrics than the Pure TFX solutions but not as good as the ones in Pure Marvin-AI solutions. The only metric that is higher in Marvin-AI + TFX solutions is the LLOC count. The reason is the code repetition needed in each step of the DASFE pipeline because the actions do not share the same variable scope. But since this is repeated code, it does not require additional effort to be created and maintained, as confirmed with the other metrics.

All these examples and the scripts to calculate the metrics are available at Apache Marvin-AI's GitHub repository. Both versions of MNIST dataset examples and MRPC dataset example with BERT are also included.

4.3 Taxonomy analysis

As discussed in Chapter 2, Lwakatare et al. (2019) researched different use case scenarios and identified five evolution stages of the use of ML components in a software project.

- The first stage is experimentation and prototyping, when the project is in the design phase, the project method of data acquisition is not adapted to ML but the sampled data prepared by specialists provide useful insights for decision making;
- Stage 2 defines the usage of ML components on non-critical deployments, when an ML pipeline is taking form but the lack of monitoring and data verification makes pipelines obscure, with considerable discrepancies between training metrics and production performance;
- Stage 3 describes critical deployments, which have a stable end-to-end pipeline that simplifies the exploration and comparison of ML models and configuration versions without affecting the end user;

- Stage 4 is achieved when the outcomes of one or more ML models are used as input to the subsequent ones. It can be efficient to user experience and reduce computational costs of the solutions, although entanglements can difficult maintenance; and
- Stage 5 achieves the maximum point of maturity, when independent pipelines with automatic testing are created. The need for human interference is minimal with tasks that would previously need this, like data annotation and monitoring. This maturity level relies strongly on quality assurance of models and data.

We performed a qualitative analysis on which of these important key features can be achieved with the help of Marvin-AI, both in the old and new architectures. The results are shown in Table 4.9.

Maturity	Key feature	Old	New
Stage 1	Feature exploration	X	X
	Simple data acquisition	X	X
	Basic model metrics	X	X
Stage 2	ML data setup	X	X
	Primitive data pipelines	X	X
Stage 3	End-to-end ML pipelines		X
	Business-centric metrics	X	X
	Experiments metadata		X
Stage 4	Sliced analysis		X
Stage 5	Quality assurance		X
	Automatic annotation		
	Pipeline independence		

Table 4.9: A qualitative analysis of Marvin-AI's old and new architectures in terms of the key features described by Lwakatare et al. (2019)

As we can see in Table 4.9, the old architecture of Apache Marvin-AI was already allowing the development of non-critical deployments. Some important features such as monitoring and a full CI/CD support described in Section 2.2 were needed to support critical deployments efficiently.

Marvin-AI has improved in terms of monitoring and prototype to deployment quality requirements with its new architecture. Thanks to Docker containerization and TFX integration, Marvin-AI's pipeline is now closer to produce solutions at the stage 4 of the taxonomy, with some stage 5 features like quality assurance of models and data, with fail safe solutions to when the minimal baseline is not being achieved. Although critical deployments were executed in Marvin-AI before, now it is easier to setup a complete environment that detects data anomalies and inconsistencies and pipeline visualization.

It is important to stress that these key features are not guaranteed by the mere adoption of Marvin-AI, but that they are made possible, or easier to achieve, with Marvin-AI. Also, the analysis demonstrated that the architectural evolution pushed Marvin-AI towards a more mature stage, as was also perceived by developers in our previous evaluations.

4.4 Benchmark suite evaluation

To evaluate Marvin-AI's new monitoring features, we performed some tests. We used two computers. For the server, we used a computer running 8 GB of RAM, Dual core Intel Core i3-5005U CPU, 7200 RPM HDD and a Realtek RTL810xE PCI Express Fast Ethernet controller. For the client, we ran *Apache JMeter* on a computer running 4 GB of RAM, Dual core Intel Core i3-5005U CPU, 7200 RPM HDD and a Realtek RTL810xE PCI Express Fast Ethernet controller. The two computers were connected in a LAN network using a Fast Ethernet network connection supported by a D-link router, model DIR-615.

We selected 9 datasets for the tests. We tried to select datasets with different features and sizes, to check how useful our approach is in diagnosing problems when using them. Table 4.10 shows the datasets used in this evaluation and the tool/algorithm that was implemented.

Dataset	Data type	# of training samples	# of test samples	# of attributes	Observations	Tool/ Algorithm
Dog Breed	Images	10222	10357	N/A	120 classes	Keras / CNN
Invasive Species	Images	2295	1531	N/A	Binary classification	Keras / CNN
Plant Seedlings	Images	4750	794	N/A	12 classes	Keras / CNN
Taxi Fare	Attribute-value	55423856	9914	7	Regression	scikit-learn / Random forest regressor
Santander Value	Attribute-value	4459	49342	4991	Regression	scikit-learn / Random forest regressor
House Prices	Attribute-value	1460	1459	79	Regression	scikit-learn / LassoCV
Santander Customer	Attribute-value	200000	200000	200	Binary classification	scikit-learn / SVM
Don't Overfit II	Attribute-value	250	19750	300	Binary classification	scikit-learn / SVM
Microsoft Malware	Attribute-value	8921483	7853253	82	Binary classification	xgboost / xgboost

Table 4.10: Datasets used in the evaluation. These can be found in `kaggle.com`

Table 4.11 shows the main results for the batch tasks. Two solutions were automatically interrupted (Microsoft Malware and Taxi Fare) due to the lack of RAM. The data shows that the maximum amount of RAM used was almost 78% for both cases. The amount of CPU used was around 100%, indicating that the chosen tool/algorithm was not making use of multiple processing cores. For these cases, additional RAM may solve the problem, as well as another algorithm to split processing in multiple cores.

Dataset	Task time (secs)	Max CPU	Max RAM	Max Disk I/O
Microsoft Malware	1082*	101%	77.7%	3.46GB / 0B
Taxi Fare	127*	98%	77.4%	2.93GB / 4.1KB
Dog Breed	600**	404%	27%	437MB / 12.3KB
Plant Seedlings	600**	397%	24%	1.79GB / 12.3KB
Invasive Species	592	390%	22%	2.79GB / 3KB
Santander Value	130	396%	7%	150,000KB / 8KB
Santander Customer	100	100%	7.5%	140,000KB / 12KB
Don't Overfit II	17	15%	0.8%	60,000KB / 15KB
House Prices	15	30%	0.8%	80,000KB / 5KB

Table 4.11: Resource usage for the batch tasks. * Automatically interrupted by the tool. ** Manually interrupted during training after 10 minutes.

Two solutions were manually interrupted after 10 minutes (Dog Breed and Plant Seedlings), as they were taking an excessive amount of time, and the estimated time for completion was several hours. For these cases, all four cores were being used at maximum capacity, as the data for maximum CPU usage shows values close to 400%. We can conclude that to reduce this time, additional CPUs or a GPU would be effective.

For the other solutions, the time varied from a few seconds (House Prices) to almost 10 minutes (Invasive Species). These times may be acceptable, but if less time is desired, additional processing power would probably reduce the time for those solutions with reported CPU values close to 400% (Invasive Species and Santander Value), as these represent the maximum allowed in this hardware. And for the solution where CPU was 100%, a different algorithm, which makes use of multiple cores, could reduce the task time.

Disk input was higher in the larger datasets, as expected, and only a small amount of disk output was noticed. For the larger datasets, a faster disk would probably help to reduce the time. In all solutions, there was no network traffic.

Table 4.12 shows the main results for the online tasks. Only the solutions for which the batch tasks were completed are shown, as there were no trained models for the others. Scenarios with 50 and 350 simultaneous threads were simulated. Each thread was configured to submit 1000 requests. For all scenarios with 50 threads there were no errors (requests without a response). The average response times were also adequate, except for the "Invasive Species" dataset, but this is expected, as this task involved processing an image in real time. Since CPU was getting close to the maximum (400%) maybe additional processing could reduce this time.

With 350 threads, all solutions had some percentage of errors, which indicate that this particular server is not able to support this amount of simultaneous requests. Again, the solution for "Invasive Species" had more errors. The maximum amounts of CPU and RAM data confirm these observations. However, in all tests, resources did not appear to reach a critical limit, therefore there are probably other limitations to this environment that is causing the errors.

# of threads	Dataset	% Errors	Ave. resp. time	Max CPU	Max RAM	Max Network I/O
50	Invasive Species	0%	1592	352%	17.1%	30MB / 30MB
	Santander Value	0%	141	246%	2.7%	93.5MB / 28.4MB
	Santander Customer	0%	147	255%	2.7%	93.6MB / 28.4MB
	Don't Overfit II	0%	115	232%	2.7%	83.5MB / 28.4MB
	House Prices	0%	79	288%	2.7%	82MB / 29MB
350	Invasive Species	4.18%	9716	352%	17.1%	238MB / 196MB
	Santander Value	1.02%	1102	254%	2.7%	691MB / 199MB
	Santander Customer	1.06%	1156	288%	2.7%	692MB / 199MB
	Don't Overfit II	1.47%	899	261%	2.7%	612MB / 199MB
	House Prices	1.87%	876	239%	2.7%	600MB / 205MB

Table 4.12: Resource usage for the online tasks

Finally, the amount of network traffic observed is proportional to the number of threads, what indicates that there is no additional network traffic other than the request/response data. These values may help to estimate cloud costs. These solutions use REST endpoints. If the costs are two high, smaller payload frameworks, such as gRPC, could be an alternative.

Chapter 5

FINAL CONSIDERATIONS

This essay describes the evolution of Apache Marvin-AI's architecture. In the end, the process of designing, implementing and evaluating Apache Marvin-AI's new architecture was successful, as it has been observed to be better than the previous one in all four independent evaluations performed. Some future improvements for Marvin-AI's architecture were also identified, such as more quality assurance techniques and more independent pipeline components. Although the new architecture partially solves many of this problems, there are still many points where manual interference is needed. There is also an absence of monitoring and debug features. Another point fairly mentioned at the feedback is the lack of updated documentation.

The quality attributes are also an indirect contribution of this process, as MLOps is a new concept and has several differences compared to traditional software development. This set of quality attributes can serve as useful guidelines to develop platforms and individual solutions that follow good development practices.

Another contribution of this work was the process we attempted to follow. Through slightly more than 8 months of research and development, from the first e-mail on 07/07/2020 until 03/18/2021 with the last architecture meeting, we departed from a completely ad-hoc process and managed to employ a simplified version of ATAM to systematically promote an architectural evolution based on quality attributes and more controlled discussions. This process also helped developers to think about the proposed changes and their impact, in a more controlled and predictable way.

5.1 Threats to validity

In this section we summarize some identified threats to the validity of our research:

TV01. ATAM is traditionally a physical meeting with different stakeholders that are invited to participate at all discussions at high level architecture specifications. Remote encounters and asynchronous discussions are not part of the main practices of ATAM.

TV02. We had to perform some serious simplifications to make ATAM work in Apache Marvin-AI's scenario. We did not compile a tree of rich scenarios with proper details and prioritization, nor promoted engaging discussions.

TV03. In the ATAM evaluation, only a few developers participated, and the results are not statistically relevant.

TV04. As the researcher is also a developer in the evaluated open source project, the qualitative analysis may be biased.

We tried to mitigate these threats as follows:

TV01. The virtual environment was required to develop the method at an open source environment. The steps, outcomes and expected discussions were made successfully even with the limitations and the architectural discussions were raised awareness at the problems and challenges.

TV02. Although the process was simplified, we managed to keep ATAM's essence, which is to use quality attributes, scenarios for the discussions and the involvement of different stakeholders.

TV03. Although we had a little amount of answers, it complied with ATAM's directive (at least five participants).

VT04. The major point of this research was to identify issues with the Apache Marvin-AI platform. As the processes involved other people and stakeholders with a diverse set of skills, the effects of this validity threat were minimized.

5.2 Lessons learned

Some important lessons were learned in this process:

- The adopted systematic process allowed the developers to identify more details about some desired changes that were not being perceived before, such as the need for a daemon to make containerization work with Marvin-AI tooling and remote development.

- Overall, we perceived that the Marvin-AI architectural evaluation has raised awareness about the importance of the architectural documentation and the participation of the community in the development work. This was also a benefit reported by others (BARBACCI et al., 2003; KAZMAN et al., 2016) in similar studies;

- Another example of a benefit of the process was the perception that Marvin-AI should prioritize usability over other attributes, because this was established as Marvin-AI's mission. This trade-off analysis was made during the discussions we made along the process;

- Similarly to what was reported by Barbacci et al. (2003), the scenarios were evaluated by Marvin-AI developers and users. Our group was aware of all design decisions regarding the architecture and contributed to them. The architectural design decisions were validated with the community;

- Although an official risk analysis step was excluded from our process, the discussions raised by the developers' feedback may have covered the need of this specific part of the scenario. The feedback also lead to the discovery of new issues with the architectural proposal and future improvements. These benefits were also observed by Barbacci et al. (2003);

- Instead of a few continuous and focused sessions of discussions and brainstorming, we relied on asynchronous communication, via e-mail, because this is how development at Marvin-AI worked. However, some synchronous meetings were necessary;

- We tried to fit as much information as possible into e-mail messages, to make communication shorter and try to get as many people participating as possible. But in some cases, we had to use other media, as it was becoming very difficult for developers to understand everything. An RFC document and video explanation helped to overcome these problems, but some had to be explained via synchronous meetings;

- Although we managed to obtain good feedback from e-mail discussions, we acknowledge that some activities are unfeasible because they require a lot of interaction and dynamism, a trivial feature in a face-to-face meeting, but difficult to organize, or even time consuming, in this form of communication;

- Defining and prioritizing scenarios was a very difficult task. We tried to organize a synchronous meeting for this purpose, but it did not go as planned, much because developers were eager to discuss more practical things instead of hypothetical scenarios. We ended up having to define them later, asynchronously, by inferring things from previous meetings and other discussions with project members.

- ATAM requires at least five participants in the discussions, and we managed to fulfill this requirement. However, not all members were the same in all meetings. We felt that a better involvement from the same members over the entire process would be better, but this was very difficult, because Marvin-AI is maintained by volunteers and we had no way to enforce one's participation in this process.

We believe these lessons can be of great value to others interested in conducting further research regarding ATAM in an open source project. We believe future work should address some limitations we encountered. First, a better way to get more people involved, from start to finish, is necessary. Another issue that needs to be better examined is an asynchronous and distributed way to define and prioritize scenarios. We faced some resistance to get developers to work on hypothetical thinking, even in an online meeting, as they were more committed with getting more things being implemented. This might also be true for the identification of the quality attributes. This was not a problem in our case, as we managed to identify them in the literature and get confirmation from developers. But in other scenarios similar barriers might be found.

Another contribution, which was not intended at first, is that this alternative to the indoor ATAM meetings is ideal for the scenario that has begun in 2020, with the Sars-CoV-2 pandemic. The simplified ATAM version can be executed in a social distancing situation with members in different geographic locations.

Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. We also thank B2W Digital, a partner company of this project, which financed and provided computational resources and data to make possible the study, through the extension activity #23112.000186/2020-97, Federal University of São Carlos.

REFERENCES

ALENEZI, M.; KHELLAH, F. Evolution impact on architecture stability in open-source projects. *International Journal of Cloud Applications and Computing (IJCAC)*, IGI Global, 2015.

ANDREWS, P. et al. Productionizing machine learning pipelines at scale. In: *Proceedings of the 32nd International Conference on Machine Learning, Lille, France*. [S.l.: s.n.], 2015.

APACHE. *Apache Beam Documentation*. Apache Software Foundation, 2021. Accessed on 2021.09.17. Available at: <https://beam.apache.org/documentation/>.

APACHE. *Marvin-AI Documentation*. MARVIN, 2021. Accessed on 2021.09.17. Available at: <https://marvin.apache.org/marvin-platform-book/SUMMARY/>.

BARBACCI, M. et al. *Using the Architecture Tradeoff Analysis Method (ATAM) to Evaluate the Software Architecture for a Product Line of Avionics Systems: A Case Study*. [S.l.], 2003.

BASS, L.; WEBER, I.; ZHU, L. *DevOps: A software architect's perspective*. [S.l.]: Addison-Wesley Professional, 2015.

BAYLOR, D. et al. Tfx: A tensorflow-based production-scale machine learning platform. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. [S.l.: s.n.], 2017.

BEHNAMGHADER, P. et al. A large-scale study of architectural evolution in open-source software systems. *Empirical Software Engineering*, Springer, 2017.

COLEMAN, D. et al. Using metrics to evaluate software system maintainability. IEEE Computer Society Press, 1994.

Deng, L. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 2012.

DEVLIN, J. et al. Bert: Pre-training of deep bidirectional transformers for language understanding. In: *Conference of the North American Chapter of the Association for Computational Linguistics*. [S.l.: s.n.], 2019.

DOCKER. *Docker Documentation*. Docker Inc., 2021. Accessed on 2021.09.17. Available at: <https://docs.docker.com/>.

FINZER, W. The data science education dilemma. *Technology Innovations in Statistics Education*, 2013.

FLAOUNAS, I. N. Beyond the technical challenges for deploying machine learning solutions in a software company. *CoRR*, 2017.

GHANTA, S. et al. {MPP}: Model performance predictor. In: *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*. [S.l.: s.n.], 2019. Accessed on 2021.09.17.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. [S.l.]: MIT Press, 2016. Http://www.deeplearningbook.org.

HERMANN, J.; BALSO, M. D. *Meet Michelangelo: Uber's Machine Learning Platform*. Uber Technologies Inc., 2017. Accessed on 2021.09.17. Available at: <https://eng.uber.com/michelangelo-machine-learning-platform>.

KAZMAN, R. et al. Evaluating the effects of architectural documentation: A case study of a large scale open source project. *IEEE Transactions on Software Engineering*, 2016.

KAZMAN, R.; KLEIN, M.; CLEMENTS, P. *ATAM: Method for Architecture Evaluation*. [S.l.], 2000.

LI, Q. et al. Medical image classification with convolutional neural network. In: IEEE. *2014 13th international conference on control automation robotics & vision (ICARCV)*. [S.l.], 2014.

LIM, J. et al. Mlop lifecycle scheme for vision-based inspection process in manufacturing. In: *2019 USENIX Conference on Operational Machine Learning (OpML 19)*. [S.l.]: USENIX Association, 2019. Accessed on 2021.09.17.

LIN, J.; RYABOY, D. Scaling big data mining infrastructure: the twitter experience. *Acm SIGKDD Explorations Newsletter*, ACM, 2013.

LWAKATARE, L. E. et al. A taxonomy of software engineering challenges for machine learning systems: An empirical investigation. In: *Agile Processes in Software Engineering and Extreme Programming*. [S.l.]: Springer International Publishing, 2019.

MARTIN, R. C. *Agile software development: principles, patterns, and practices*. [S.l.]: Prentice Hall, 2002.

NGUYEN, G. et al. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, Springer, 2019.

PROMETHEUS. *From metrics to insight: Power your metrics and alerting with a leading open-source monitoring solution.* 2021. Available at (accessed jun 2021): https://prometheus.io/.

SCULLEY, D. et al. Hidden technical debt in machine learning systems. In: *Advances in neural information processing systems*. [S.l.: s.n.], 2015.

SERBAN, A. et al. Adoption and effects of software engineering best practices in machine learning. *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020.

SILVA, L. C. et al. Benchmarking machine learning solutions in production. In: *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*. [S.l.: s.n.], 2020.

SQLITE Documentation. SQLite Community. Accessed on 2021.09.17. Available at: <https://www.sqlite.org/docs.html>.

SRIDHAR, V. et al. Model governance: Reducing the anarchy of production ML. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. [S.l.]: USENIX Association, 2018.

Staples, M.; Zhu, L.; Grundy, J. Continuous validation for data analytics systems. In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. [S.l.: s.n.], 2016.

TENSORFLOW. *Tensorflow RFC process*. Alphabet Inc., 2021. Accessed on 2021.09.17. Available at: <`https://www.tensorflow.org/community/contribute/rfc_process`>.

TFX. *Tensorflow Extended Documentation*. Alphabet Inc., 2021. Accessed on 2021.09.17. Available at: <`https://www.tensorflow.org/tfx/api_overview`>.

ZHOU, J. et al. Katib: A distributed general automl platform on kubernetes. In: *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*. [S.l.: s.n.], 2019.