Cloud-based web application for multivariate time series analysis

A language-agnostic integrative architecture for short- and long-running machine learning algorithms

Vasco de Campos Candeias

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Alexandra Sofia Martins de Carvalho
Prof. Susana de Almeida Mendes Vinga Martins

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. Alexandra Sofia Martins de Carvalho
Member of the Committee: Prof. João Nuno de Oliveira e Silva

January 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfils all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgements

This thesis marks the end of five years of study, a journey that I could have never undertaken alone.

As such, I express my sincere gratitude to my supervisors, Professor Alexandra Carvalho and Professor Susana Vinga, for their invaluable knowledge, relentless guidance and never-ending patience during this atypical year.

A word of appreciation is also warranted to Professor Jonas Almeida, whose indispensable input and thought-provoking conversations paved the way for elaborating and implementing the architecture hereby presented. Moreover, I would like to acknowledge Professor Pedro Adão, whose expertise allowed to guarantee the security of the developed system.

To my parents, a heartfelt thank you: it would not have been possible to get here without you. I extend this acknowledgement to my partner, family and friends, who have continuously been there for me, even in the most troublesome times.

In particular, I wish to thank my dear friend Luz Rocha for her unwavering support and help on finding the voice I needed to defend this thesis.

Furthermore, I would like to recognise Instituto de Telecomunicações and Fundação para a Ciência e a Tecnologia (FCT) for respectively assigning and financing the grant, in the scope of project PREDICT (PTDC/CCI-CIF/29877/2017), that supported this research.

Last but not least, I would like to thank the reader for their interest in my dissertation.

Resumo

O crescimento de registos de séries temporais multivariadas em biomedicina tem impulsionado o desenvolvimento de algoritmos para a sua clara interpretação. As redes de Bayes dinâmicas estão entre os métodos mais populares para analisar este tipo de dados, por facilitarem a compreensão das suposições subjacentes. Contudo, o desenvolvimento e utilização de programas relacionados com estas metodologias não espelham o forte progresso teórico que se tem verificado nesta área. De facto, numa era em que inúmeros serviços estão já disponíveis na Internet, é natural que os investigadores e profissionais de saúde estejam relutantes quanto a executar estas aplicações computacionalmente dispendiosas na linha de comandos dos seus computadores. Nesta tese, desenvolveu-se uma aplicação *web*, disponível em `https://vascocandeias.github.io/maestro`, que preenche esta lacuna, graças à sua capacidade de executar estes algoritmos exigentes e incorporar qualquer tipo de *software* programável para análise de dados. A ferramenta proposta conta com uma arquitetura de microsserviços, implementada na nuvem, que comprovadamente lhe permite aumentar a capacidade de forma instantânea, reduzindo, assim, os tempos de execução. A sua escalabilidade é, também, garantida, não se tendo verificado qualquer degradação de desempenho quando a aplicação foi submetida a milhares de pedidos concorrentes. Além disso, a sua versatilidade é demonstrada através do estudo de dados reais. É, ainda, distribuído em `https://github.com/vascocandeias/maestro-backend` um servidor local idêntico que pode ser usado numa rede interna, para que os dados dos utilizadores nunca saiam de uma rede restrita. Estas ferramentas deverão permitir a fácil e rápida utilização dos mais recentes métodos para análise de dados longitudinais.

Palavras-chave: aplicação *web*, arquitetura na nuvem, MAESTRO, redes de Bayes dinâmicas, séries temporais multivariadas

Abstract

The surge of multivariate time series records in biomedicine has driven researchers to develop algorithms towards their clear interpretation. Dynamic Bayesian networks are among the most popular methods, allowing to transparently impute, classify, and make predictions on medical records while enabling users to understand the underlying assumptions. Although significant theoretical progress has been made on these methodologies, the development of related programs and their adoption is still far from fully accomplished. Indeed, these packages remain as resource-intensive command-line applications that analysts are reluctant to use. The lack of a public web application that integrates these methods is crucial in an era where every service is quickly moving to the Internet. In this thesis, this missing website is conceptualised and implemented as a cloud-based application that comfortably scales with the number of requests and can be easily extended with any scriptable software for data analysis. MAESTRO (dynaMic bAyESian neTwoRks Online), available at `https://vascocandeias.github.io/maestro`, relies on a microservice architecture deployed on Amazon Web Services and can handle the most demanding tasks, with the flexibility to readily increase processing power and reduce execution times. This application's scalability is proven by making thousands of simultaneous calls, which does not cause any performance degradation. Its versatility is further conveyed through a case study with real data. Additionally, a local server to be used in an intranet, preventing data from leaving a managed network, is proposed and distributed at `https://github.com/vascocandeias/maestro-backend`. These tools should allow clinicians and investigators to effortlessly use state-of-the-art tools for longitudinal data analysis.

Keywords: cloud architecture, dynamic Bayesian networks, MAESTRO, multivariate time series, web application

Contents

List of Tables

List of Figures

Acronyms

AES Advanced Encryption Standard.

API Application programming interface.

AWS Amazon Web Services.

BASE Basically available, soft state and eventually consistent.

bcDBN BFS-consistent k-graph dynamic Bayesian network.

BN Bayesian network.

CaaS Container as a service.

cDBN Consistent k-graph dynamic Bayesian network.

CSV Comma-separated values.

DBN Dynamic Bayesian network.

EC2 Elastic Compute Cloud.

EMR Electronic medical record.

EQF Equal frequency binning.

EQW Equal width binning.

FaaS Function as a service.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

IaaS Infrastructure as a service.

IAM Identity and Access Management.

JAR Java Archive.

JSON JavaScript Object Notation.

JWT JSON Web Token.

LOCF Last observation carried forward.

LR Linear regression.

MAESTRO Dynamic Bayesian networks online.

ML Machine learning.

MTS Multivariate time series.

PaaS Platform as a service.

PGM Probabilistic graphical model.

QoS Quality of service.

REST Representational state transfer.

RPC Remote procedure calls.

S3 Simple Storage Service.

SDK Software development kit.

sdtDBN tDBN with static and dynamic variables.

SES Simple Email Service.

SOAP Simple object access protocol.

SQS Simple Queue Service.

tDBN Tree-augmented dynamic Bayesian network.

TS Time series.

UDDI Universal Description, Discovery, and Integration.

URL Uniform Resource Locator.

VM Virtual machine.

W3C World Wide Web Consortium.

WSDL Web Services Description Language.

WWW World Wide Web.

XML Extensible Markup Language.

Chapter 1

Introduction

1.1 Motivation

Despite the increased development of machine learning (ML) techniques for automatically diagnosing human diseases from electronic medical records (EMRs) [1, 2], there is still a lack of accessibility for the end-user. In a world where every service is moving towards the web and people are less willing to download and execute software on their devices, it is imperial to provide a web-based platform that integrates the state-of-the-art methods for issues such as imputation and clustering of multivariate time series (MTS). This system would allow such analyses to be readily available for any practitioner and let them take full advantage of the increased availability of EMRs, which hold the patients' clinical evaluations, and hence originate MTS when collected over time.

Since doctors cannot just put their patients' health in the hands of black-box solutions [3], they need models that not only can handle problems with a vast number of attributes and extensive data sets but are also transparent and meet the required level of accountability of the medical field [4]. By resorting to probabilistic graphical models (PGMs), these experts may get interpretable tools that produce reliable predictions and facilitate efficient interpretation, allowing them to improve each inference over their patients' EMRs.

As follows, it is no surprise that a subset of PGMs, the dynamic Bayesian networks (DBNs), are emerging in this field, being used for tasks such as detecting outliers in MTS [5], making predictions using these time series (TS) or grouping them into clusters [6]. As such, it would be highly beneficial for the medical community to have a platform at their disposal that implements these already available packages in a way that allows the understanding of the underlying algorithms and assumptions.

To fill in this gap, we studied state-of-the-art implementations of web applications and developed MAESTRO (dynaMic bAyESian neTwoRks Online) [7]. This publicly available web tool aggregates imputation, outlier detection, clustering and inference tools based on DBNs as well as discretisation and visualisation capabilities that allow users to upload and analyse their MTS. Along with the minimisation of the implementation cost, the resulting architecture should be aimed at maximising the following characteristics:

- Quality of service (QoS) – How well the system performs from the user's perspective;
- Security – How the system guarantees integrity, confidentiality and availability;
- Horizontal scalability – How easy it is to add nodes to accommodate the increase of users;
- Vertical scalability – Whether it is possible to add resources to improve performance;
- Concurrency – Whether the resources can be accessed by multiple users simultaneously;
- Extensibility – How easy it is to add new functionalities;
- Transparency – Whether the user perceives the system as a whole or as distributed procedures;
- Longevity – How long the web tool will remain online.

One of these requirements is particularly challenging when trying to serve these exponentially complex algorithms to learn DBNs online: concurrency. In fact, the one web application that makes use of these models for outlier detection – METEOR [8] – exhibits severe concurrency issues. When a user is training a network, the entire website becomes unavailable to others, even the landing page. As such, a crucial objective is ensuring that not only does the website continue to operate but that simultaneous requests will not influence one another.

Besides, while it is clear that other data-analysis software like SPSS [9] already exist, they often require licenses that might not be affordable and do not allow research or medical teams to add custom packages. Furthermore, even though some tools such as Weka [10] may seem to satisfy the requirements mentioned above for being open source and extensible, they still require users to download software and have limited extensibility by only accepting Java packages. Moreover, having to run the program locally is not just an inconvenience, as practitioners might not even have access to computers with the necessary hardware to rapidly make these computations and without sacrificing performance on other vital tasks.

In contrast, MAESTRO is more than a freely available service. By downloading its source code, any medical or academic institution can have this modular microservice infrastructure running in their intranet by solely setting some environment variables and executing a single command. Moreover, adding custom packages has never been easier, as the developers only need to create two straightforward interface files and copy them, along with the executables, to the server's directory.

To face the challenges mentioned above, MAESTRO relies on a microservice architecture with a gateway that receives the users' requests, a message queue where workers get tasks from and a data service to store input files and results. On top of these, an email service notifies analysts upon each task's completion, a log service tracks potential errors, and an authenticator ensures authorised access.

While an on-premises version of the web tool is also made available, the publicly available one – which is the focus of this thesis – is deployed in the cloud using Amazon Web Services (AWS) and tested for its concurrency and scalability. We will verify whether, by resorting to both serverless functions and dynamically allocating virtual machines (VMs), the system withstands bursts of thousands of simultaneous requests without any performance degradation. Likewise, we intend to prove that this implementation can vertically scale for improved computational power and reduced execution times.

1.2 Contributions

This dissertation culminated in the following contributions:

- Comprehensive review of Internet-based systems architectures;
- Conception of an extensible system architecture for data analysis using DBNs;
- Deployment of the devised application as a widely available cloud-based web tool, accessible at `https://vascocandeias.github.io/maestro`;
- Publication of a video tutorial that exemplifies the use of the application, which can be consulted at `https://youtu.be/VWfPkxaSWMI`;
- Demonstration of the implemented functionalities through a case study using real Alzheimer's disease data;
- Performance evaluation of the publicly available tool by stress testing it with ambitious requests;
- Implementation of an on-premises version of the application, distributed at `https://github.com/vascocandeias/maestro-backend`;
- Proposal of possible future work to improve the architecture and its implementations.

The aforementioned have also been compiled in an article which is awaiting approval by the co-authors to be submitted to the international journal *Computer Methods and Programs in Biomedicine* [11].

1.3 Document outline

Having introduced the problem to solve in this thesis, this chapter is followed by an introduction to the relevance of ML in biomedicine as well as a review of the state-of-the-art approaches for Internet-based system architectures in Chapter 2.

A more explicit definition of the problem at hand is then devised in Chapter 3, where its requirements are drawn from a set of imposed constraints. The proposed functionalities are then explored in Chapter 4, followed by the system design in Chapter 5.

Having explained the architecture, the two implementations of this application are presented in Chapter 6. In Section 6.1, we go through its deployment using a cloud infrastructure, and in Section 6.2, we analyse its on-premises version.

Chapter 7 begins with a demonstration of using the web tool in Section 7.1 and further includes some stress testing in Section 7.2 that allows to benchmark the implementation's scalability and concurrency. A discussion regarding these findings is then made in Section 7.3.

Finally, Chapter 8 serves as a wrap-up of this dissertation, summarising its most significant aspects and suggesting future work that can be undertaken to improve this tool.

Chapter 2

Background

In this chapter, we start by analysing the relevance of ML, and the use of DBNs in specific, in biomedicine. Then, we look into some technical details relevant for any computer engineer when planning to develop a web tool. Since these applications may run on either the server or the client's browser, these paradigms are separated into two different sections.

2.1 Machine learning in biomedicine

2.1.1 The rise of electronic medical records

Health providers have for long collected medical data from their patients and resorted to knowledge about the general population to learn how to treat them better [12]. But until very recently, these documents only consisted of paper records. However, their digitalisation, allied with fast-growing computational resources, pave the way to saving lives and helping treat diseases that are so far considered incurable.

In 1995, Hersh predicted a heavy interaction between clinicians and computers [13]. He was not wrong, and 25 years later, the European Bioinformatics Institute revealed that by the end of 2019 they had accumulated 307 petabytes of raw medical data, an increase of 34 over the course of a year, and an average of 62.6 million daily requests [14].

Naturally, it follows that the ML field has contributed immensely to the medical community, with autism subtyping [15] and lymph node metastases detection from breast pathology [16] being two notable cases. However, it still faces many obstacles when seeking to make use of the medical data that is gathered daily [17].

Therefore, if in 1995 there was the fear that the adoption of electronic medical records was not following the growth of computer technology, the tables have now turned, and practitioners are struggling to find intelligible methods to analyse them. This need led to the surge of multiple ML models such as the Bayesian networks (BNs).

2.1.2 Bayesian networks

When dealing with ML models in biomedicine, one common concern is to have a model that not only works but is also interpretable. This is where PGMs come into play, with BNs being one of its major branches. To illustrate how these directed acyclic graphs are built, a simple example should suffice. Take an individual that has a tumour. While its type probability – whether it is malignant or benign – might depend on its size [18, 19], it is definitely not related to the person's Social Security number (SSN). This situation can be represented by the BN in Figure 2.1 without any loss of insight.

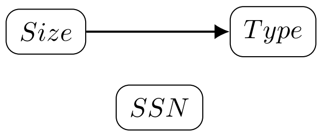


Figure 2.1: Example of a simple Bayesian network.

In a BN, the nodes represent variables and the edges their dependencies. Taking Figure 2.1 as a case in point, the edge means that the probability of the type of tumour being malignant or benign given its size, $P(Type \mid Size)$, is not just the type probability, $P(Type)$. At the same time, the SSN has, in no way, an impact over the type of tumour. As such, there is no link between these two edges, and $P(Type \mid SSN) = P(Type)$. Hence, this model is probabilistic and not deterministic in nature.

Even though this example is intuitive and easily deductible by hand – after all, it is clear that a tumour's size may have an impact on its type [18, 19] –, this is not the case for real-world applications of these models, and, as such, many insights can be deduced by simply computing a BN and visually analysing it.

2.1.3 Dynamic Bayesian networks

Although the traditional BNs provide helpful insight regarding the relationship between attributes, they lack the accounting for time and cannot depict this critical dimension for medical forecasting. This is where DBNs are needed so that doctors can track the evolution of a patient over time and analyse how some measurements might affect future outcomes.

Taking Figure 2.2 as an example, the relevance of this model for the study of MTS becomes clear: it allows to analyse how multiple variables relate themselves between two points in time. For instance, the type of tumour will influence the next therapy session, which, in turn, will have an impact on the size. Notwithstanding, the latter only dictates the tumour type at the same time point.

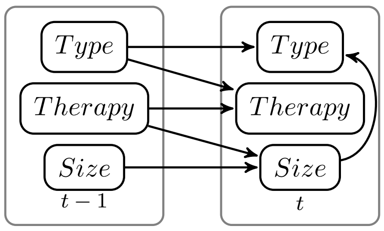


Figure 2.2: Illustration of a dynamic Bayesian network.

However, notice how only the previous instance of time is taken into account. In the given case, the idea that causality goes beyond one time slice is evident, as the size of a tumour in a moment t does not solely depend on its size at $t-1$, as it might be growing or shrinking. This apparent negligence of information is the application of the Markov property: a forgetting assumption.

By taking one step back and generalising the attributes to $\boldsymbol{X}$, let $\boldsymbol{X}^{(0:t)}$ be their state from time 0 to t, and $\boldsymbol{X}^{(t+1)}$ the state at time $t+1$. Naturally, the probability of these variables from time 0 to T, $P\left(\boldsymbol{X}^{(0:T)}\right)$, will be given by

$$P\left(\boldsymbol{X}^{(0:T)}\right) = P\left(\boldsymbol{X}^{(0)}\right) \prod_{t=0}^{T-1} P\left(\boldsymbol{X}^{(t+1)} \,\middle|\, \boldsymbol{X}^{(0:t)}\right). \tag{2.1}$$

Since this is too complex to compute, the Markov assumption states that there is an equivalent equation,

$$P\left(\boldsymbol{X}^{(0:T)}\right) = P\left(\boldsymbol{X}^{(0)}\right) \prod_{t=0}^{T-1} P\left(\boldsymbol{X}^{(t+1)} \,\middle|\, \boldsymbol{X}^{(t)}\right), \tag{2.2}$$

by assuming that attributes in the next step are independent of their values in the past if their present state is known, that is, if

$$\left(\boldsymbol{X}^{(t+1)} \perp\!\!\!\perp \boldsymbol{X}^{(0:t-1)} \,\middle|\, \boldsymbol{X}^{(t)}\right) \tag{2.3}$$

holds.

While this might erroneously suggest that variations in previous time slices, such as the one suggested, should not be analysed, note how it just means that a new attribute should be added to the network: the growth rate, for this scenario. Consequently, the network in Figure 2.2 turns into Figure 2.3. Nonetheless, there might always be some loss of information or context, and that is why there is a relaxation to this assumption: the network can include time-slices up until $t-m$, where m is the Markov lag.

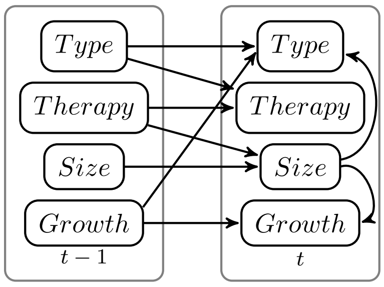


Figure 2.3: Example of a dynamic Bayesian network.

Finally, time invariance might also be assumed. This is the case when a network that models the relationship between two time slices can be generalised to accurately represent any other consecutive observations. These networks are called stationary and, in other words, assume that the dynamics of the system do not change over time. Transposing into the running example, this would mean that a tumour that has expanded with only one therapy session and another that has been growing over multiple sessions – two very different scenarios – are the same. To accommodate this assumption, another attribute can be added – Duration of Therapy –, and the model becomes applicable to any moment in time.

2.2 Server-side processing

Web applications are traditionally envisioned as client-server relations where a front-end service makes requests to a back-end server, which processes the heavy tasks. When compared to client-side processing, where the requests never leave the user's browser, this paradigm leverages the server's computational power, which should be far superior. Moreover, the extensibility of these environments is endless since there are no limits on what to install on the servers.

Due to its overwhelming scope, this approach poses questions such as where to offload the processes to, how the components, or even the browser and the servers, communicate with each other, or how to ensure security. All of these topics are covered in this section.

2.2.1 Computation offloading

The purpose of processing the data in a server is to offload the computation to an external platform so that the user's resources are not consumed and allowing the tasks to be rapidly concluded. This can be done through various approaches, as explained below.

Cluster computing The most conventional offloading consists of multiple servers performing similar tasks and therefore ensuring availability and fault resilience through redundancy. Furthermore, node replication allows for easier horizontal scaling by server acquisition instead of relying entirely on vertical scaling by hardware replacement. This architecture has been successfully used in data analysis, being the foundation of the data imputation web application WIMP [20].

Grid computing Once considered the future of global computation [21], this type of offloading is similar to cluster computing in that it uses multiple nodes but differs on how each of them operates. While in clusters every server handles the same application, grid computing nodes perform distinct tasks and are more loosely coupled, being adequate for embarrassingly parallel problems, i.e., work that requires minimal manipulation to be separated into parallel tasks.

Cloud computing The most recent offloading paradigm is defined as a system that provides on-demand computing power, storage and other services from a shared pool of computing resources [22]. This paradigm is therefore specialised in distributed computing and has the edge over the so-far introduced models when it comes to scalability, abstraction and maintenance [23].

Edge computing Since the cloud might pose some privacy concerns and have higher latency, edge computing brings the data storage and computation closer to the user [24, 25]. By placing the edge servers nearby, bandwidth usage reduces dramatically, and organisations no longer have to expose their data to others. Building on this idea, a centralised service may provide the methods to be used by edge servers for processing, which will upload less, if any, information back to the main one. However, to ensure data safety, this paradigm requires the organisation to have dedicated servers suited for the needed computation, which may be complicated for some applications and institutions.

2.2.2 Monoliths and microservices

Servers have been commonly used to deploy monolithic systems – structures that have their entire logic as a single application. While this is the most comfortable method of quickly releasing a viable product, it no longer satisfies the scalability and rapid development required by modern big companies. Microservices – the decomposition of an application into a set of loosely coupled services – have thus emerged as a growing architectural style that attempts to tackle these issues.

Developing this newer type of architecture is more complex, and so is deploying features that comprise multiple services, since these are distributed systems [26]. In contrast, improving each functionality becomes a continuous and fast process, due to the underlying separation of concerns [27].

As far as more objective comparisons go, studies have shown that microservices may reduce costs dramatically while improving performance, albeit with increased latency [28, 29]. This does not come as a surprise, since each request will have to take more steps to be handled. However, the scalability advantages of this architecture might compensate for this shortcoming, as it is much easier to increment the number of processing nodes: it is an inherent characteristic of this pattern, whereas monolithic services require load balancers to scale.

All in all, microservices should be seen as a gradual migration from monoliths, where components can be more or less loosely coupled. If decomposing the architecture into multiple services from day one seems straightforward, then it may be the best course of action. Otherwise, starting with a working monolithic platform and progressively decomposing it into distinct services might be the best approach. After all, this was the path followed by companies like Netflix and Amazon [30].

2.2.3 Synchronous communication

Given that this approach relies on a client separated from the server, it is necessary to establish how the communication between them will be done. Moreover, when opting for a microservices architecture, multiple components in the back-end of the application are separated and thus require establishing connections. As such, this section focuses on detailing various methods of synchronous communication, where a client makes a request and waits for the server's reply.

Remote procedure calls (RPC) and simple object access protocol (SOAP)

RPC is the most "native" approach of the two, having been first implemented in 1984 [31]. This paradigm allows remotely calling procedures as if they are local while hiding the entire process from the user, being a significant step forward for the area of distributed systems. In fact, all the client needs is the interface through which the server offers its operations, providing seamless client-server computing. Its required components can be distinguished in Figure 2.4, where it is clear that the client must request the procedure from a *stub* (the process that converts the parameters passed between the two entities) which, through the *communication module*, requests the server to run it. Finally, the latter's *dispatcher* selects the appropriate *stub* for the received procedure identifier, who can then unmarshal the parameters, have the server procedure run, and marshal the output, which is similarly returned to the client.

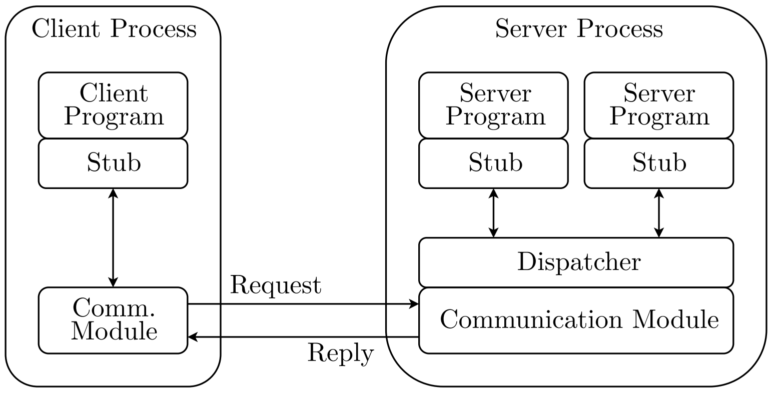


Figure 2.4: RPC architecture.

At the same time, to accommodate object-oriented programming, Remote Method Invocation was born [32]. The most significant change was allowing to call a method in a remote object: something impossible with RPC. However, this application programming interface (API) is Java dependent, meaning that it only works for communications between processes developed in this language.

However, even if being widely adopted in its early stages, RPC has also been subject to a lot of criticism when it comes to using it across a network due to issues related to transparency, crash handling, and lack of parallelism [33]. To improve upon this technology, many protocols emerged, with one of them being the simple object access protocol (SOAP) [34].

SOAP appears as the successor of XML-RPC – the RPC for the Web – and is able to communicate over more than just Hypertext Transfer Protocol (HTTP) and Hypertext Transfer Protocol Secure (HTTPS), supports a broader range of security implementations and, above all else, allows the use of Web Services Description Language (WSDL). The latter is an Extensible Markup Language (XML) document published by the service provider to a Universal Description, Discovery, and Integration (UDDI) registry and describes the server interfaces such as its procedures and data types. The client only has to find this interface in the service registry and then communicate directly with the service provider via XML, already knowing its interfaces. This process is illustrated in Figure 2.5.

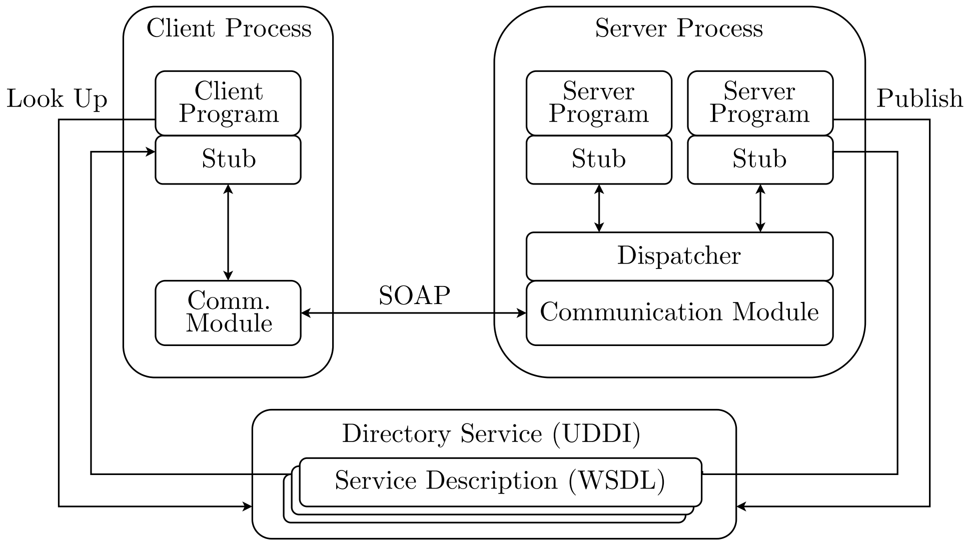


Figure 2.5: SOAP architecture.

Representational state transfer (REST)

More recently, Web Services evolved into a paradigm of resources, where every resource has its own interpretable Uniform Resource Identifier (URI) and is therefore accessed hierarchically. For instance, `/students/vasco` represents one object while `/students` describes an entire collection of this type of objects. To access and manipulate these resources, Fielding proposed the REST architecture in his famous dissertation [35] with the central feature of adopting the HTTP uniform interface, whose methods are:

- `GET` – To access existing resources;
- `POST` – To create a new resource;
- `PUT` – To modify existing resources;
- `DELETE` – To delete existing resources.

In his dissertation, Fielding purposes that this new architecture should move as much load away from the server and towards the client as possible. As such, the first constraint imposed on his original style was adopting the client-server separation of concerns by isolating the user interface from the data storage. This simplifies the server and consequently not only improves scalability but also allows components of the system to evolve separately: when a server is updated to a newer version, the client has no need to change anything, with the update going unnoticed.

Secondly, the stateless concern is a significant game-changer in this new architecture. The server no longer maintains records of the session of a user and receives the needed information from the client. A simple example would be that of a website with multiple pages that should be rendered sequentially. In a stateful design, the server has the information that the user is currently viewing the first page. Hence, the user only needs to ask for the next page and will get the second one. Converting this scenario into a stateless design, the server does not need to keep in memory the user's current page, as the latter provides both this information and a request for the next one. In other words, the server does not maintain state, the user does. Nonetheless, there are some trade-offs. For starters, there is a noticeable increment in the amount of data being transferred over the network, as the client is now the one keeping the state and must provide the server with every information needed to complete a given request. This might lead to a reduction of bandwidth and loss of packets (and their consequent re-transmission). Furthermore, the correct application development is much more dependent on the client's implementation, which might lead to some inconsistent behaviour.

Another constraint that improves scalability, but now reducing the network congestion and improving efficiency, is caching in the client by setting the resources as cacheable when sending them from the server. This directly heightens user satisfaction as it dramatically reduces latency, since a lot of requests are no longer needed.

Finally, REST also improves on extensibility by allowing mobile code to be run on demand: a client might get the code needed to perform a specific action and execute it himself instead of overloading the server unnecessarily. Again, this significantly improves scalability and reduces network congestion.

Comparison

As can be seen from the review of literature, multiple arguments can be made in favour of each paradigm. Many comparisons have been made as well [36, 37], and the consensus is that none is clearly better than the other.

For instance, while REST is easier to develop and maintain on the client-side, SOAP is more suited for distributed programming, as it is built on top of RPC. Moreover, on the one hand, when using SOAP, many errors will be caught when compiling the code, while as for REST, this detection will be delayed until execution. On the other hand, being restricted to such generic operations means that it is much easier to make changes to the method implementations since the coupling between client and server is much looser.

Overall, while REST outperforms in execution speed, scalability, compatibility, and simplicity, SOAP shines when it comes to security and reliability [36, 37]. As such, an adequate approach would be to build a small prototype using both protocols and compare them. This approach led to the discovery that, even if still being supported, SOAP has been deprecated from Java and is no longer included by default since its 11th version. While searching about this decision, the consensus was that developers should simply switch over to REST unless some SOAP-specific features were absolutely needed. This prompted further research on the modern trends for Web Services, with some intriguing results being found on Google Trends – Figure 2.6.

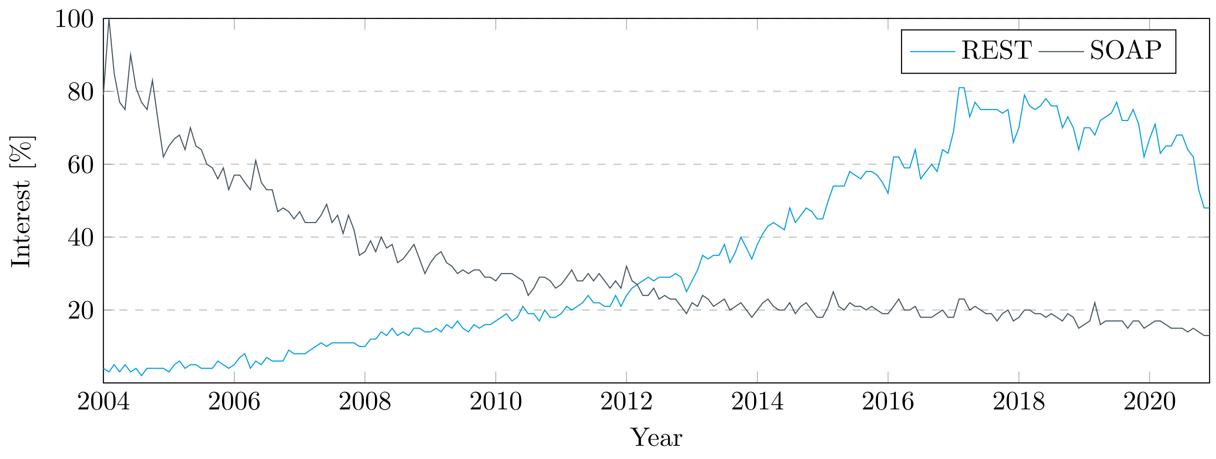


Figure 2.6: Worldwide Google searches for “Representational state transfer (kind of software)” and “SOAP (protocol)” from January 2004 until December 2020. Data source: Google Trends (`https://www.google.com/trends`). Consulted in 23rd December 2020.

Even though Google searches do not dictate that a particular technology is better than the other, they do indicate how the interest over each of them has progressed over time. As such, it is interesting to see that SOAP has been losing interest since 2004, whereas REST has been on the rise, which goes in line with some published articles that stood by the latter’s adoption [38, 39]. In addition, these results support the data presented by Halili and Ramadani [37], who further claim that most new projects should adopt the REST architecture.

2.2.4 Asynchronous communication

Synchronous communication may sometimes be inadequate, as services cannot always block waiting for replies. As such, there is a need for asynchronous methods of delegating tasks, a problem which this messaging pattern tries to solve [26].

In essence, this pattern uses a channel to where clients post messages, which will be read by servers. Doing so relies on the assumption that the former does not need to receive a reply to his request immediately, as long as this is sent at some point in the future. In fact, he may not even require an answer at all, splitting this pattern into two main paradigms:

- Request/reply – The client will receive a response to his request, albeit asynchronously;
- Publish/subscribe – Subscribers will receive and interpret the messages but will not reply to them.

The first one might seem like an unnecessary complication of synchronous communication. Still, it brings many benefits, such as improved scalability – as any number of servers can be listening for requests – and easier loose coupling – as the client must only know to which channel he should address the messages. Furthermore, it makes it simpler for the client to attend to other tasks while waiting for the response, as he will get a notification of this message instead of having to interrupt his logic. However, message queues are not bi-directional, so this approach has the increased complexity of dealing with a channel for each communication direction, as the server must reply to a different one.

On the other hand, publish/subscribe channels allow the client to have one-to-many relations, where any number of subscribers receives the published message and processes it. A widespread use of this architecture is to have the client notifying every subscriber of changes in the state of an application. Another vastly implemented workflow is having many processing nodes available but only one of them attending to each message, assuring its completion. Here, the message broker may also act as a load balancer, fairly distributing the messages without overwhelming any node.

Even though asynchronous communication usually implies a more complex architecture, messaging is one of the most used patterns in microservices, seeing that clients are often satisfied by knowing that less urgent tasks will be completed whenever possible.

2.2.5 Databases

No system is complete without data storage, where processes perform create, read, update and delete (CRUD) operations. Nowadays, these break down into relational and NoSQL databases.

Relational Dating back to the 1970s, relational databases were developed to provide atomicity, consistency, isolation and durability (ACID) guarantees, in an era where data was small, and there was no need to have distributed data stores [40]. In modern days, they benefit from features such as transactions, which improve consistency, or allowing complex queries with table joins, thus remaining the most well-known and widely adopted databases. As such, they are every developer's entry point to the realm of data storage.

NoSQL With the growth of big data analysis and cloud computing, there has been an urge to develop databases that can instantly scale horizontally by being simpler and distributed over clusters of computers. However, the famous CAP theorem clearly states that from consistency, availability and partition tolerance, only two may be chosen [41]. This was not a problem that relational databases faced, since these were originally centralised systems. For distributed databases, however, either consistency or availability must be, even if lightly, left out, with NoSQL solutions opting to sacrifice the former and introducing the ironic basically available, soft state and eventually consistent (BASE) properties.

Comparison When compared, while relational databases allow transactions and perform better for complex queries that join tables, NoSQL stores have the upper hand as far as overall performance and scalability go, since they are based on key-value systems and can be easily sharded and deployed in multiple nodes [42, 43].

2.2.6 Security

When offloading sensitive data analysis to external servers, it is mandatory to ensure privacy by taking authorisation and confidentiality measures. Since HTTPS [44] already secures data in transit, we should focus on understanding how to provide authentication and security at rest.

Authentication and authorisation

Traditionally, authentication in web applications has been achieved using sessions and cookies: the user sends his username and password to the server, who replies with a session cookie. From then on, every request is sent with this cookie, and the server is responsible for matching it with the client's information. This solution, however, does not scale well, since different server replicas would need to have access to the same table, originating a point of failure and tight coupling.

More recently, JSON Web Tokens (JWTs) were introduced [45] as a way to transparently share the client's information between both agents: the server replies to the authentication request with the required information for future use. If this is attached to subsequent requests (usually in the HTTP headers), the servers have instant access to the user details. By properly encapsulating the information and signing the JWT, the server also ensures it cannot be tampered with and that no impostor can use his own token to confuse the back-end into thinking he is someone else.

However, an intruder who could somehow get this token would be able to impersonate another user and access the latter's sensitive data. To secure against these attacks, the OAuth2 specification [46] states that this access token must expire shortly, recommending a lifetime of 10 minutes. This does not mean the client must resend his credentials regularly. Instead, by also receiving a refresh token with a longer lifespan, the browser can check if the access token has expired before each request and, if needed, refresh it using this other secret.

Finally, having safeguarded authentication, it is then up to the resources servers to determine which content the authenticated user has access to.

Confidentiality

When data privacy is a concern, encryption is the first suggestion to emerge. Using the Advanced Encryption Standard (AES) with 256 bits is generally considered as safe as possible, and it is even trusted by the U.S. government [47] for securing the country's most secret information. Therefore, encrypting data at rest might seem to be enough to ensure privacy.

The problem is when services need to use the data, temporarily decrypting and loading it into memory, a period during which a proficient software engineer may access the memory pages and retrieve the data. The most straightforward solution is to encrypt the volume where the application is running itself. This way, it would be necessary to bypass the software security during execution to retrieve the information, as it could only be read from within.

Some might say this still poses a security risk since developers might have backdoors [48] for troubleshooting that may be exploited. And they are right. For this reason, security specialists are now recurring to homomorphic encryption [49], which allows operations to be performed on top of encrypted information, meaning that the client is the only one to see both the decrypted input and output. The downsides are that, as of the time of writing, these methods require specific programming libraries, do not encompass every computational operation and are overall too slow [50].

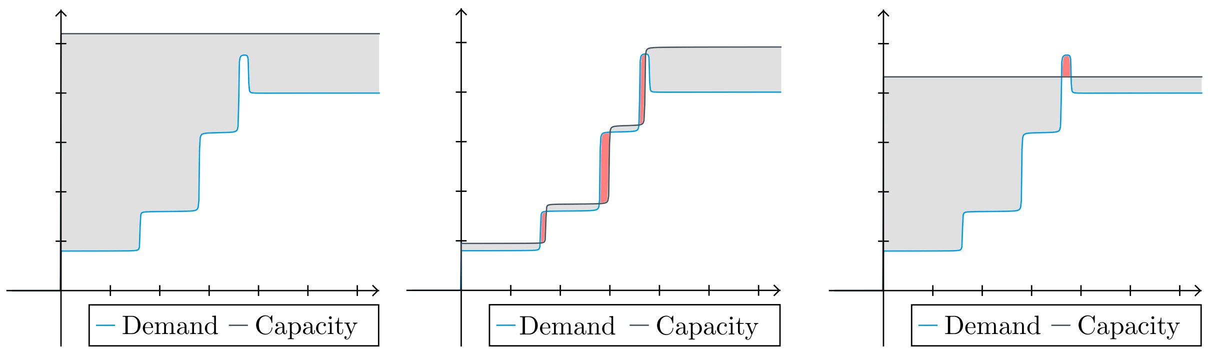
2.2.7 Deployment

When opting for a server-side processing system, it is necessary to establish where to host it, with the more traditional approach being to acquire a server to deploy the services. This can be done in multiple ways, each bringing a new layer of abstraction [26].

Firstly, by deploying code directly on the server, its resources may be more efficiently used, but there is no isolation between different services and their consumption cannot be controlled. Secondly, by using virtual machines (VMs) – computer systems emulations that allow replication across various hosts –, services can be isolated by encapsulating the entire technology stack. Nevertheless, these deployments are relatively slow, as building these images is not immediate, and depend upon system administrators keeping the operating systems updated. Finally, the developer may use containers – lightweight images that merely contain applications while sharing the host's kernel –, which have improved portability and deployment speed but still need system and runtime patching.

What is common to all of these approaches, however, is that they require a premature prediction of the workload; that is, it is necessary to anticipate how many users the system will have and what the size of their data will be. This leads to the situations in Figure 2.7.

The best solution in terms of meeting demand is the one depicted in Figure 2.7a, where the prediction was well done, and the resources are enough to provide the service. In this scenario, the peak load is accounted for, and the demand is never higher than the capacity. However, notice the size of the shaded area: this represents economic losses. If the server is idle most of the time because the resources were allocated for the peak, it means that the server is consuming power unnecessarily. This leads to the scenario in Figure 2.7b, where the capacity grows with the demand. This increase might be challenging



(a) Perfect provisioning. (b) Capacity adjustment according to demand. (c) Peak disregard.

Figure 2.7: Demand and capacity scenarios over time: grey areas represent unused excess, red areas represent insufficient capacity.

to follow, as upgrading the server cannot be done instantaneously and, in the worst case, may even require a complete replacement. In fact, this approach requires a system administrator to keep track of the demand, predict its growth, and be ready to scale the system whenever needed. Finally, it is also possible to disregard the peaks – Figure 2.7c – and try to serve the customer most of the time, having a worse performance whenever the demand is higher than the capacity. This solution might be more cost-effective but definitely does not provide the best QoS.

All of the solutions presented above have their limitations and one notable drawback in common: they require a system administrator that keeps track of the demand. This intuitively means that a project which is not the main focus of an organisation would end up being disconnected, as the server would eventually be “forgotten”. This is where looking into cloud options may be rewarding.

Infrastructure as a service (IaaS)

Popularised by Amazon in 2006 [51], this cloud service model consists in renting spare hardware to clients, with minimal maintenance conducted by the provider. It is up to the service developer to update, patch and secure it, while the IaaS provider solely ensures that the hardware remains operational [52].

Paired with auto-scaling and a load balancer, these services can scale up and down automatically – albeit not instantaneously –, keeping up with the client’s needs. Additionally, since clients have access to the entire VMs, they gain deeper control over security and performance.

Container as a service (CaaS)

If containers follow VMs in local deployment, the same is true for the cloud. This newest layer of abstraction allows clients to develop their applications using any environment they require since the latter is described and replicated in a server without the developer having to deal with the underlying resources [53].

This technology facilitates testing – since the local environment is similar to the production one –, deployment – given the container images’ portability – and scalability – as the cloud provider can easily replicate the described container [54].

Platform as a service (PaaS)

When the user is comfortable with outsourcing the server maintenance and runtime definition, simply wanting control over the application and its data, choosing this technology might be the answer [55].

In a nutshell, platform as a service consists in deploying long-running services, similarly to IaaS, but without having to set up the entire server and exclusively focusing on code development, with the cloud provider tending to the remaining matters such as network, operating system or storage.

Nonetheless, to ensure high availability and allow long-running tasks, these applications do not allow scaling down to zero without some considerable cold start upon relaunching, negatively affecting response times in applications like web servers. On top of that, since there is no control over the operating system and the installed languages and libraries, the developer may not have every framework he needs at his disposal.

Function as a service (FaaS)

Amazon led the way again by releasing its FaaS solution in 2014 [56], being the first large public cloud provider to allow code to scale up from zero instantly and down again with minimal cold start – the time it takes to launch the first instance in a new host.

These functions can be seen as short-lived scripts that run in response to triggers such as file uploads or HTTP requests and focus on the user paying only for the execution time, with no provisioning. Withal, this is not to say these functions are flawless, as they do have some shortcomings – from limited language support to time execution caps – that prevent their adoption in every cloud application.

Serverless

Serverless computing is a server solution provided by big organisations, such as Google or Amazon, who, by having many costumers, can balance the load between their servers, optimise their use, and rapidly supply them to clients.

On the one hand, serverless computing allows scaling the application up and down as fast as possible. In other words, the resources are made available on-demand, as soon as they are needed. On the other hand, when the demand is lower, so is the capacity, implying that there are no wasted resources. This reduces the need for provisioning and allows the programmer to focus on the service itself. Furthermore, the service provider only pays for when the service is in use, instead of having to pay for idle times or having a fixed price per period.

Moreover, there is no need for a system administrator anymore: the provider automatically manages services' resources. This is why this concept and FaaS are commonly used interchangeably, even though serverless file storage or authentication have existed for years [57]. However, this misconception is now changing with the appearance of CaaS, which aim at being serverless containers [53].

As far as drawbacks go, while some are directly related to which "as a service" is running on top of this execution model, others are inherent to the technology: latency is forcefully higher than for already running instances and state is harder to maintain [57].

2.3 Client-side processing

As already stated, the lesser the load on the server, the higher the scalability of the service. Ergo why not move the entire process to the client and have the service run in the user's browser? This can be achieved with code on demand, which, as always, has its pros and cons.

JavaScript

In conjunction with Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS), JavaScript forms the trio of core technologies of the World Wide Web (WWW), allowing scripting in webpages. Even though this language was initially intended for use in the client-side, it has grown to be so powerful that its server-side counterpart – Node.js – has been, since 2015, the language whose package manager has more modules by a large amount – Figure 2.8.

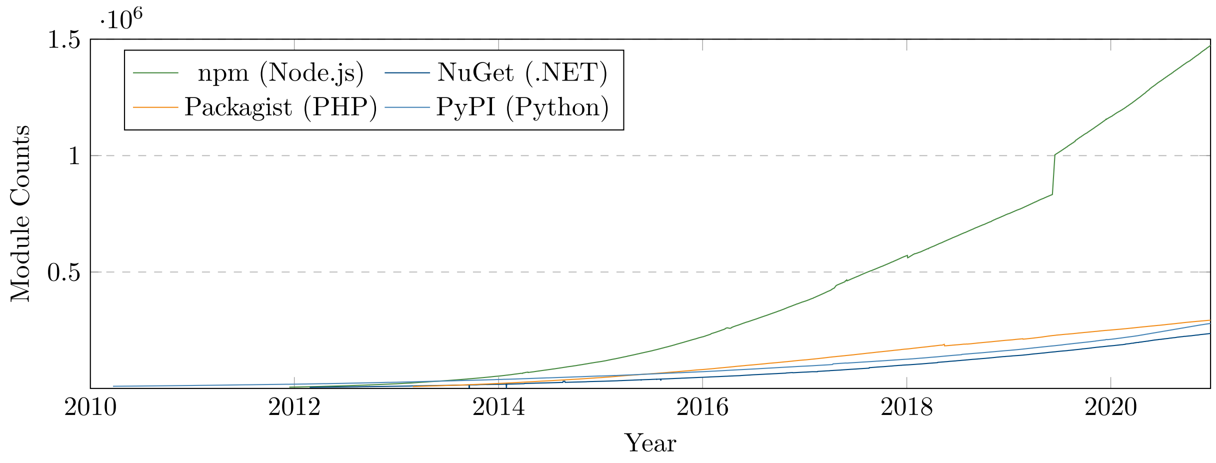


Figure 2.8: Number of modules for some representative languages from March 2010 until December 2020. Data source: Module Counts (`http://www.modulecounts.com`). Consulted in 23rd December 2020.

As far as performance is concerned, JavaScript has been tested for running deep learning algorithms on the browser, using the Tensorflow.js library, and the results are comparable to the native frameworks when taking advantage of the computer's GPU [58]. Nevertheless, it is also stated in this research that deep learning on the web is still in its early stages and thus has a lot of room for improvement.

WebAssembly

> *The arrival of WebAssembly expands the range of applications that can be achieved by simply using Open Web Platform technologies. In a world where machine learning and Artificial Intelligence become more and more common, it is important to enable high-performance applications on the Web, without compromising the safety of the users,* [59]
> – Philippe Le Hégaret, W3C Project Lead

On the 5th December 2019, the WWW trio becomes a quartet when the World Wide Web Consortium (W3C) officially recommends WebAssembly as the fourth language for the web in a press release [59] and by now every major web browser already supports it.

But what is WebAssembly? Wasp, as it is abbreviated, defines a new portable binary code, allowing to bridge the gap between strongly typed languages and the web. Doing so enables programs written in languages such as C/C++ or Rust to run on the client's browser. And the advantages do not stop there: research has shown that regardless of still being in its first version, WebAssembly can achieve benchmark results close to native C [60] – a truly remarkable feat.

Finally, a significant drawback of this approach is the current lack of native support for Java to Wasp compiling. This can be bypassed by using compilers such as JWebAssembly or TeaVM, but this may lead to errors that cannot be yet fixed.

Comparison

While both have proven to be capable of high-performance computation, experimental results corroborate the notion that WebAssembly surpasses JavaScript, as it shows noticeable gains over the latter [60]. Moreover, the former does not require any transpilation, as the source files can be compiled into .wasm files. However, this technology is still in its early stages and, as such, its support might not be sufficient.

All in all, one thing is certain: both of these client-side approaches have the privacy advantage on their side, as there is no need for users to upload sensitive information to the server, and that is something that the server-side solutions cannot surpass.

Chapter 3

Problem definition

Having covered the essential theory behind designing and implementing an Internet system, it is now paramount to clearly establish the web tool's objectives and constraints, which will then be used to define its fundamental requirements.

3.1 Constraints

Since the resulting website is meant to remain freely available and considering it is not a major product of any organisation, its budget is necessarily modest, leading to one of the most restrictive constraints: finding a low-cost solution that safeguards its longevity. However, opting for an economical design and implementation should not mean sacrificing security, since medical data entails a high level of confidentiality and privacy for which data anonymisation is important but insufficient.

Moreover, there is one other decisive characteristic of the website's functionalities to take into account: time complexity. Most methods hereby used to learn DBN structures are linear in the number of observations, polynomial in the number of attributes, and exponential in the number of parents [61–63], meaning that computing a network with four parents is not just twice as time-consuming as a network with two, but takes exponentially more time instead. As such, the analyses' execution times are not bounded, either above or below, meaning they can take seconds, hours, or even days. Consequently, users might want to be notified whenever prolonged requests have finished.

On top of this, we must further comprehend the target users: researchers and practitioners. Due to the nature of these fields of knowledge, these professionals must be able to extend the tool's functionalities with their own programs, in spite of the programming language and environment. Also, they should at any time have access to their data sets – uploaded as potentially sizeable CSV files – and previous analyses, whose processing time cannot depend on the server's load, seeing that the programs are already inherently slow.

Last but not least, it is essential that deploying the platform is as straightforward as possible, allowing any hospital or investigation centre to adopt and modify it to better suit their needs. This completes the set of major constraints, which can be summed up in the following topics:

- The solution ought to be low-cost for improved longevity;
- Medical data needs to be secured;
- Tasks may be both short- and long-running;
- Researchers and practitioners need to be notified when their results are finished;
- Execution time cannot depend on the server's load;
- Every data set and past analysis must be immediately available;
- Users may need to add their own packages, despite the development language;
- Sizeable CSV data sets may be uploaded;
- Deployment must be as simple as possible.

3.2 Requirements

After gathering the constraints, these can be converted into a list of musts to keep in mind during the designing and implementation phases:

- **The system should lean towards free or low-cost solutions**, contributing to its longevity;
- **The entire infrastructure must be encrypted** to protect sensitive data;
- **The application must tolerate both short- and long-running tasks**, to accommodate exponentially complex running times;
- **The architecture ought to be extensible** to integrate new packages supplied by analysts;
- **The server must be vertically scalable** to ensure that processing more demanding tasks in less time is possible, should it be needed;
- **The system is required to scale horizontally**, accommodating userbase growth;
- **The design must ensure concurrency** so that users are not affected by others' requests;
- **The web tool needs an authentication system** that allows users to access their private data sets and previous analyses;
- **The tool should contain a notification system** that alerts users when results are ready;
- **The services ought to gracefully handle errors** and clearly present them instead of having the packages quit unexpectedly without any output, leaving it to the user to interpret the issue and adjust the input;
- **The system is required to ensure transparency**, with users perceiving it as a whole;
- **The tool ought to accept data sets of any size**, as MTS records may be sizeable;
- **The back-end must be developed with service orchestration tools** to facilitate deployment.

Chapter 4

Functionalities

As previously stated, this web tool should encompass many packages useful for data analysis. These follow the methodological pipeline depicted in Figure 4.1 and will be briefly presented in this section, along with their input parameters and files.

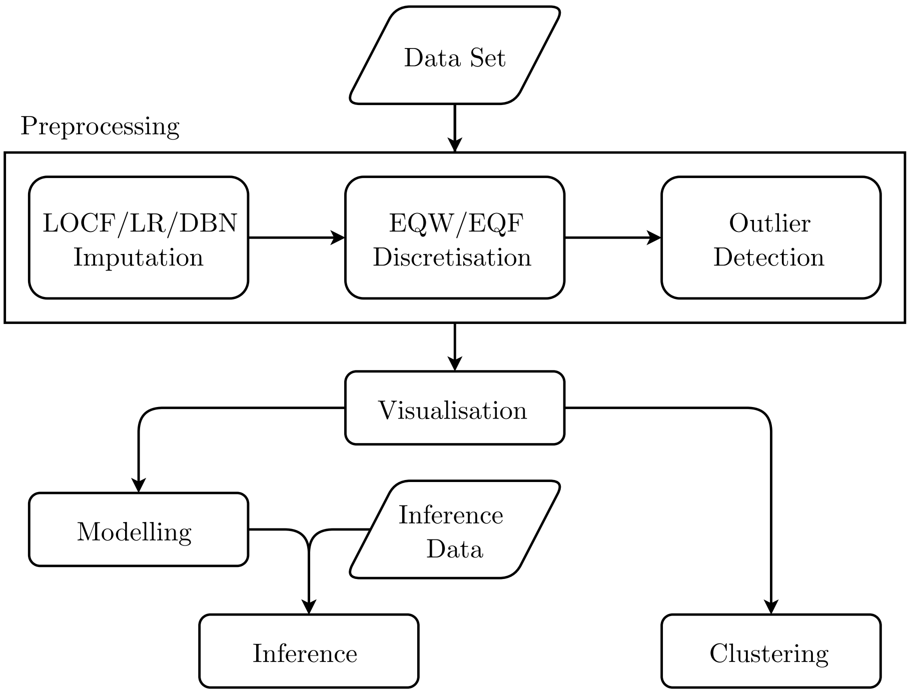


Figure 4.1: Web tool pipeline for analysing data sets.

4.1 DBN learning

Before diving into the proposed functionalities, a brief overview of methods for learning DBNs is warranted, as most packages use them. It is thus pertinent to clarify that finding these models is NP-hard, that is, it is at least as hard to solve as any nondeterministic polynomial time problem, meaning that there must be some kind of relaxation. Hence, this model selection challenge is often approached with score-based learning algorithms, i.e., methods that choose the best network according to some given score.

Tree-augmented dynamic Bayesian network (tDBN) [61] This temporal model assumes intra-slice dependencies, whereas most methods in the literature focus on inter-slice relations only. The search space is then limited by restricting the former to tree-augmented networks, meaning that nodes have at most one intra-slice parent. Unlike the heuristic algorithms, this model has the advantage of finding a global maximum instead of a local one.

BFS-consistent k-graph dynamic Bayesian network (bcDBN) [62] To exponentially increase the tDBN's search space in the number of variables, the bcDBN algorithm starts by computing an optimal tDBN. It then looks for dependencies consistent with the breadth-first search (BFS) order of this tDBN, that is, relations that are in the transitive closure [64] of this order.

Consistent k-graph dynamic Bayesian network (cDBN) [63] This method is similar to the latter but does not use the level-order, opting for a simple consistency with the topological order induced by an optimal tDBN.

tDBN with static and dynamic variables (sdtDBN) [65] While the others only accept temporal features, this algorithm also takes static ones. Moreover, it is possible to specify prior knowledge, i.e., constraints on which attribute relations are mandatory or forbidden, thus restricting the search space.

4.2 Data preprocessing

As a first step, it is necessary to preprocess the given data before trying to learn from it, since most methods require discrete and complete data. Here, it is possible to receive both continuous and discrete data. For the latter, discretisation is skipped, while it is mandatory to discretise any continuous input.

4.2.1 Imputation

Since most method implementations assume the MTS are complete, the given data either must not have any missing values, or these should be imputed using one of the following methods:

- **Linear regression (LR)** – Generates a linear interpolation using the available values in the TS and assigns the missing values accordingly;
- **Last observation carried forward (LOCF)** – Copies the most recent state prior to the missing one. If there is none, the next value is used;
- **learnDBN** [66] – Learns a tDBN, bcDBN or cDBN despite the missing values and then uses it to impute them.

When the input is continuous, the first two methods can both be used, with the last two being viable otherwise. For this step, the first two simple methods were developed using Python and have the following input argument:

- **-i** *(string)* – Name of the data set.

In contrast, the `learnDBN` Java package [66] takes multiple parameters:

- **-bcDBN** *(Boolean)* – Learn a bcDBN structure;
- **-cDBN** *(Boolean)* – Learn a cDBN structure;
- **-i** *(string)* – Name of the data set;
- **-ind** *(number)* – Intra-slice in-degree;
- **-m** *(number)* – Markov lag;
- **-ns** *(Boolean)* – Learn a non-stationary network;
- **-p** *(number)* – Number of parents;
- **-sp** *(Boolean)* – Force intra-slice connectivity to be a tree.

4.2.2 Discretisation

After correcting for missing data, it is paramount to discretise it as the proposed models are only suited for this sort of input, which can be done by either one of the following:

- **Equal frequency binning (EQF)** – Creates bins containing approximately the same number of data points;
- **Equal width binning (EQW)** – Splits the range of values into a fixed number of bins and respectively distributes the data points.

As in the simple imputation methods, the discretisation packages were developed using Python, taking the following arguments:

- **-d** *(string)* – Discrete attributes space separated;
- **-i** *(string)* – Name of the data set;
- **-n** *(number)* – Number of bins.

It is important to acknowledge that both of these methods take not only the number of bins but also a parameter with a string containing the discrete features. This ensures that the package will not reduce already discrete attributes to smaller alphabets.

4.2.3 Outlier detection

The final step of preprocessing the input is to detect its outliers, that is, both the transitions (from time-slice t to $t+1$) and the subjects (i.e., entire time series) that considerably deviate from the rest. This phase replicates METEOR [5], which learns a tDBN and then scores each transition and TS. By setting a score threshold, it is then possible to classify them as being legitimate or outliers.

Considering the transitions cannot be removed from the input file, it is only possible to automatically filter out entire subjects to continue to the next step. In any case, the results can also be downloaded as a CSV file with the classification or score of each transition or subject.

Unlike `learnDBN`, this package allows scoring with both minimum description length (MDL) and log-likelihood (LL) criteria, as can be seen in the following arguments:

- **-i** *(string)* – Name of the data set;
- **-ll** *(Boolean)* – Use LL scoring criterion;
- **-m** *(number)* – Markov lag;
- **-mdl** *(Boolean)* – Use MDL scoring criterion;
- **-ns** *(Boolean)* – Learn a non-stationary network;
- **-p** *(number)* – Number of parents.

After retrieving the network and scores from the back-end, METEOR requests the servers every time the user adjusts the threshold or number of bins in the histogram. This should in no circumstances be done, since JavaScript is perfectly capable of handling these calculations and renderings. Doing so results in the user having immediate feedback instead of waiting for replies.

4.3 Visualisation

Although in Figure 4.1 this step appears after preprocessing the data, it is also used to show the data to the user in a graphical way at any other stage, so that the evolution of the attributes is evident. This plot, inspired by the R package `mvtsplot` [67], may be accomplished using the D3.js JavaScript framework.

Since a data set of multiple MTS has four dimensions (time, subject, value and attribute), we may display a single feature per diagram, allowing users to compare subjects over the same metric.

4.4 Analysis

Having preprocessed the file, the user can decide whether he wants to split the data into clusters of subjects or make a prediction based on new input. While the later relies on separately generating a DBN model that fits the data, this is not the case for the first one.

4.4.1 Clustering

Using the program developed by Arcadinho [68], it is possible to group subjects in clusters. To determine them, the author proposes finding the k DBNs which best describe k clusters of subjects in the data, and then classify each TS accordingly [6]. Besides downloading the clustering results, the user may analyse the learnt networks – which can be tDBNs, cDBNs or bcDBNs – to understand the model that could generate each cluster. This method's arguments are as follows:

- **-bcDBN** *(Boolean)* – Learn a bcDBN structure;
- **-cDBN** *(Boolean)* – Learn a cDBN structure;
- **-i** *(string)* – Name of the data set;
- **-ind** *(number)* – Intra-slice in-degree;
- **-k** *(number)* – Number of clusters;
- **-m** *(number)* – Markov lag;
- **-ns** *(Boolean)* – Learn a non-stationary network;
- **-p** *(number)* – Number of parents;
- **-sp** *(Boolean)* – Force intra-slice connectivity to be a tree.

4.4.2 Modelling

Before making inference on new data, it is necessary to model a DBN using the `sdtDBN` package [65], which allows learning an sdtDBN structure. By including the mandatory and forbidden dependencies, clinicians may specify relations between attributes that they are certain exist and exclude those they already know to be impossible, which might effectively save time when facing model selection problems. The resulting model is then available to be later used for inference and saved for future use. This packages accepts the following parameters:

- **-b** *(number)* – Number of static parents;
- **-i** *(string)* – Dynamic features to be used for network learning;
- **-is** *(string)* – Static features to be used for network learning;
- **-m** *(number)* – Markov lag;
- **-mA_dynPast** *(string)* – Dynamic nodes from $t' < t$ that must be parents of each $X_i[t]$;
- **-mA_dynSame** *(string)* – Dynamic nodes from t that must be parents of each $X_i[t]$;
- **-mA_static** *(string)* – Static nodes that must be parents of each $X_i[t]$;
- **-mNotA_dynPast** *(string)* – Dynamic nodes from $t' < t$ that cannot be parents of each $X_i[t]$;
- **-mNotA_dynSame** *(string)* – Dynamic nodes from t that cannot be parents of each $X_i[t]$;
- **-mNotA_static** *(string)* – Static nodes that cannot be parents of each $X_i[t]$;
- **-ns** *(Boolean)* – Learn a non-stationary network;
- **-p** *(number)* – Number of dynamic parents;
- **-sp** *(Boolean)* – Force intra-slice connectivity to be a tree.

4.4.3 Inference

Using the already learnt model, new observation files and another functionality of the `sdtDBN` package, it is possible to make two sorts of inference: either predict how the new time series will progress until a certain time point or infer a given attribute in specific time slices.

This step allows, for both prediction and inference, two types of sampling modes:

- **mostProb** – The most probable values are always chosen;
- **distrSampl** – The values are randomly sampled according to their probability distributions.

When inferring, it is also possible to view the probability distribution, with intermediate values being chosen using **distrib**, as can be seen in its arguments:

- **-fromFile** *(string)* – Serialised object of the sdtDBN;
- **-inf** *(string)* – Variables to perform inference on;
- **-infFmt** *(string)* – Inference format: **mostProb**, **distrSampl** or **distrib**;
- **-obs** *(string)* – Dynamic observations to make inference;
- **-obsStatic** *(string)* – Static observations to make inference.

As for prediction, its parameters are as follows:

- **-fromFile** *(string)* – Serialised object of the sdtDBN;
- **-infFmt** *(string)* – Inference format: **mostProb** or **distrSampl**;
- **-obs** *(string)* – Dynamic observations to make inference;
- **-obsStatic** *(string)* – Static observations to make inference;
- **-t** *(string)* – Final timestamp.

Chapter 5

Architecture

Before implementing the web tool, a careful design of its architecture should be elaborated. To do so, this chapter will serve as a description and comparison of some proposed architectures, which will allow selecting the most suitable approach.

5.1 Possible architectures

After reviewing the literature and going through the system's requirements, it was possible to define several possible architectures: two client-side and one server-side.

The difference between the first two is merely writing the code in JavaScript or compiling it to WebAssembly. To host these applications, a free and reliable option would be GitHub Pages [69], since both run on the browser. If the tools were to be developed from scratch, JavaScript would be the best approach: it is a mature language that has proven to be able to handle demanding machine learning tasks, as shown in Section 2.3. However, having the Java programs already been developed, transpiling to JavaScript might be a step back when it is possible to compile them to WebAssembly. Nevertheless, this is such a new technology that there is still a lack of official support for Java and, as such, it should not be taken for granted.

On the other hand, the imputation, discretisation, clustering or any other package can be used as it is and serve as a microservice behind a gateway that decides which of them to request for each step of the pipeline. The reasoning behind having multiple microservices instead of running everything in a monolithic back-end is to separate concerns and improve scalability and extensibility, with new services being easily added to the system. For this architecture, both SOAP and REST were considered as synchronous communication protocols, and a comparison between the two can be read in Section 2.2.3. With such evidence of REST's growth in relative popularity hereby presented and having no concrete reasons to choose the oldest of the two, the chosen paradigm for this thesis was the representational state transfer. The main advantage of server-side processing is the ease of adaptation and release of the already available packages whereas its most significant disadvantage is the user having to upload the (possibly sensible) data to a server. To sum up, this architecture is depicted in Figure 5.1.

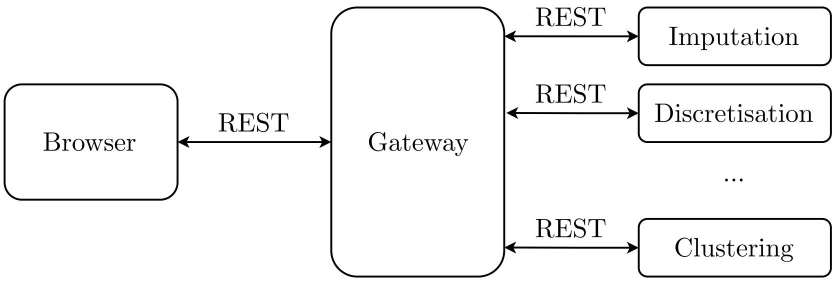


Figure 5.1: Server-side architecture.

5.2 Early prototype

Given both the advantages and disadvantages of each architecture, an early server-side model was developed, since this was the most comfortable architecture to prototype rapidly. This early version, which mostly serves as a proof of concept, was a minimal implementation of the clustering package [68] and is represented in Figure 5.2, where the Python interface calls the Java Archive (JAR) as it is, passing in the requested parameters and specifying the multi-threading flag. The purpose of this mock-up was to illustrate and analyse some of this architecture's advantages, facilitating a more educated decision.

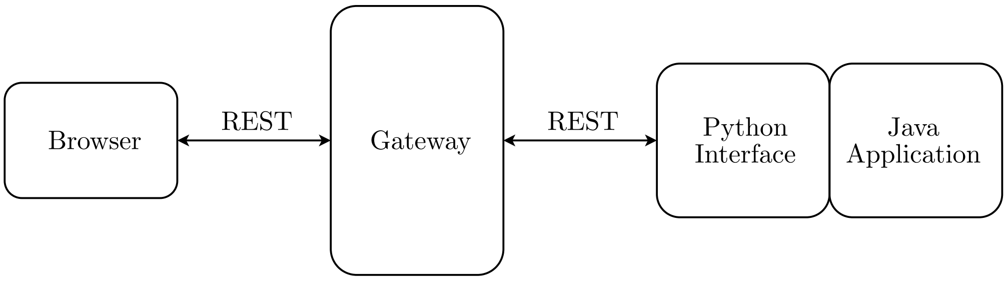


Figure 5.2: Prototype architecture.

As far as implementation goes, both the gateway and the interface are simple Python 3 scripts running the Flask web framework, with the first rendering the HTML templates itself. However, to reduce network and server load, in a final version, the browser should only retrieve the results and dynamically compose the webpage accordingly.

The implemented clustering service, running on the root endpoint of the system's `5001` port, receives a CSV file with the data set as well as a JSON body with the following keys:

- **type** *(string)* – Which network should be trained: tDBN, cDBN or bcDBN;
- **-ns** *(string)*– Indicating if there should be one transition network per time transition (non-stationary network). If so, its value should be equal to `checked`;
- **-pm** *(string)* – `checked` if the network parameters should be learned and outputted;
- **-sp** *(string)* – To force the intra-slice connectivity to be a tree instead of a forest, eventually producing a structure with a lower score, this should be `checked`;
- **-ind** *(number)*– The in-degree of the intra-slice network;

- **-k** *(number)* – Number of clusters in the data;
- **-m** *(number)* – Maximum Markov lag to be considere. If not specified, it will be equal to 1;
- **-p** *(number)* – Maximum number of parents from preceding time-slices, which is also 1 if not specified.

As for the tests, they were done with the `combinedDataset.csv`, available in the package's GitHub Page [68] and consisting of 2,000 TS with five attributes and 10 time steps each, produced by two distinct DBNs. Moreover, two different computers were used:

- Laptop 1 – Intel i7-10710U, with 6 cores, base frequency of 1.10 GHz and a max turbo frequency of 4.70 GHz. Also used as the server;
- Laptop 2 – Intel i7-5500U, with 2 cores, base frequency of 2.40 GHz and a max turbo frequency of 3.00 GHz.

For each computer, two tests with two clusters, multi-threading enabled and every remaining parameter set to default were done: one executing the program locally and another using the prototype running on Laptop 1.

The results were as expected: not only did the program keep its correctness, but, as Table 5.1 suggests, the elapsed time of the web tool for Laptop 1 was approximately the same as when executing the JAR file locally. Moreover, while Laptop 2 took over two times as long as the other to run the program locally, when using the prototype, it took less than 10 % more time. This residual difference could be explained by Laptop 1 using the loopback interface, avoiding some network delay, but what should be noticed, however, is that Laptop 2 runtime goes down to almost half since it is taking advantage of the server's resources. Furthermore, this could only improve using JavaScript in the browser to request the response instead of having the back-end rendering the HTML.

Table 5.1: Time elapsed (s) when clustering the `combinedDataset.csv` five times for each laptop and taking the average (A) and standard deviation (σ), both locally and using the prototype, with the default parameters, two clusters and multi-threading.

Client	Locally		Prototype	
	A [s]	σ [s]	A [s]	σ [s]
Laptop 1	1.42	0.18	1.49	0.15
Laptop 2	3.17	0.35	1.60	0.07

Secondly, another test, whose results are shown in Table 5.2, also revealed a possible complication: by tweaking the parameters, the analysis took around 52 seconds in the best scenario and 98 in the worst. Requests to REST APIs should return almost immediately so that the browser does not hang and freeze. As such, this is a major factor to take into consideration when designing the architecture.

Other than that, given the standard deviations, the reason for the prototype running faster than locally for Laptop 1 in this second test might be the result of natural fluctuations. Furthermore, Laptop 2 has once again shown a considerable gain from using the server, reinforcing some disadvantages of client-side processing, especially when the latter is limited to running in one single thread.

Table 5.2: Time elapsed (s) when clustering the `combinedDataset.csv` five times for each laptop and taking the average (A) and standard deviation (σ), both locally and using the prototype, with a bcDBN, an intra-slice in-degree of two, a maximum of three parents, two clusters and multi-threading.

Client	Locally		Prototype	
	A [s]	σ [s]	A [s]	σ [s]
Laptop 1	55.15	15.15	51.79	10.83
Laptop 2	97.96	19.93	61.75	6.62

5.3 Choosing an architecture

When choosing the architecture for a web application, it is essential to consider the input characteristics and the complexity of the tasks to be executed. By doing so, it is possible to put aside the possibilities that would not fulfil the basic requirements. As such, Chapter 3 should now be revisited.

For this system, no matter the package, the input is well defined: each program receives a small and constant number of parameters and up to six (depends on the package) data sets, represented as CSV files, for which the size is unknown: they can either be small (a few hundred bytes) or large (hundreds of megabytes), depending on the number of attributes, subjects and time points.

If the time discrepancy between distinct requests had been theorised in Section 3.2, it has now been demonstrated in Section 5.2, since learning a network with three times the number of parents took 50 times longer. Dealing with these time complexities results in apparently small adjustments having a hefty impact on the execution time, leading to another unknown: the programs might take just a few seconds to run, but may also take many minutes, even hours or days, to complete.

All in all, the fundamental requirements to take into consideration at this point are:

- The input files are in a text format – CSV;
- The size of the input files can go from hundreds of bytes to hundreds of megabytes;
- The execution time has no upper or lower bounds.

Regarding the first observation, the fact that the input can be either sent in files or the body of the request (as it is already pure text) gives considerable flexibility on how to handle the requests, therefore alleviating any possible restrictions on both the client- and server-side architectures.

The same can be said for the second conclusion, as the size of the input files has no impact on choosing whether the application runs on the server or client-side since the size of the uploaded files has no limit on either: both allow gigabytes of data to be used. In fact, as far as the server architecture goes, the HTTP protocol does not establish any maximum request size, meaning that this value is only determined by the browser and server used, with the most popular solutions allowing the mentioned multiple gigabytes of file upload.

However, this is no longer true when it comes to the execution time. If a client-side website takes too long to execute a request, it is the users' experience that suffers, as they will usually not wait for hours to get a result while having their computer potentially slowed down from the data analysis.

And this only gets worse when a user wants to analyse multiple data sets at the same time, as the work is accumulated over each running request. Furthermore, while it would be possible to admit that long-running requests could be preformed overnight, the consequence would be to have a service that is only truly useful if the user does not intend to use the computer simultaneously, and that is prone to an accidental power outage deleting any progress on the requested analysis.

On top of it all, the client's resources are always limited, and multiple requests would run side by side, limiting each other's computing power and, consequently, prolonging the execution. In opposition, a server can always not only be improved, providing more power and, potentially, faster results but also scaled, as already mentioned in Section 2.2.7.

Taking the above facts into consideration, the client-side architecture must be put aside, as it would provide the worse QoS, reliability and concurrency (from a single user perspective).

5.4 Improving the server-side architecture

Knowing about the program's potential time complexity, exposed by the prototype implemented in Section 5.2, the solution schematised in Figure 5.1 is not viable either, since neither should the browser indefinitely hang waiting for the response nor should the gateway keep stacking up pending requests. This hints at running the packages asynchronously, in the background, and allowing the user to retrieve the answer when it is ready, without having to keep the browser tab open. In other words, the gateway should not communicate with these microservices via REST, as this implies a fast response, but delegate the request and answer the client immediately.

Building on this idea, and considering Section 2.2.4, using a publish/subscribe message queue perfectly fits this scenario. Replacing the synchronous requests to the methods services with this channel allows the gateway to place a message in the queue and immediately reply to the client, while a worker will read and handle the pending task. Moreover, this solution will improve scalability, given that when more computing power is needed to distribute requests, instead of buying a new and more powerful server to move the services into, all that is necessary is to run a new worker in another server. This way, the old one is not wasted, and the computing power may be easily doubled, with the message queue handling the load balancing.

After replacing the last layer of the back-end with the proposed communication channel, the architecture evolves into the one depicted in Figure 5.3.

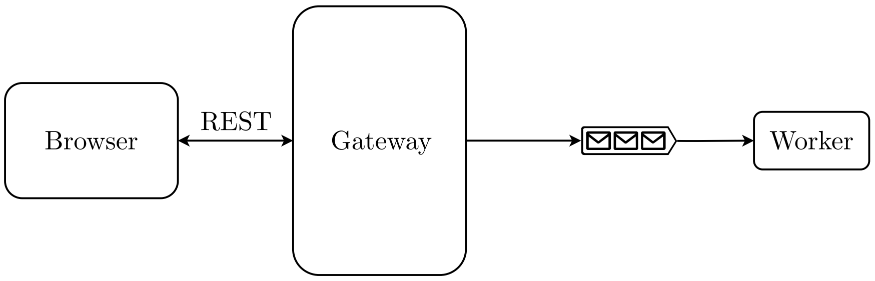


Figure 5.3: Server architecture with the message queue and a worker.

Notice how the multiple microservices so far considered are now replaced by a single one, the worker, which can be replicated across multiple servers to increase the number of processing nodes. Although a point could be made that each package should be its own microservice for improved extensibility, this level of abstraction is still done inside the worker, as will become apparent in Section 5.5. Moreover, when thinking about scaling the system, having one microservice per package would require one process to be running per method, implying that whenever the available resources needed to be increased, the number of new microservices to be launched would be equal to the number of packages. At the same time, this would rely on the assumption that every package is requested with the same frequency, when in fact some might rarely be needed, but their processes would still be running, consuming system resources.

Having defined the workflow from the gateway to the workers, it is now necessary to establish how the results will be sent back to the user. In fact, the browser is no longer waiting for any response, considering it has already received an answer from the gateway informing that the request is being executed. With that in mind, the information that the gateway sends back should be enough to allow the browser to query the back-end for the result, which can be done either periodically, until the worker has finished, or sporadically, whenever the user needs to access the information. This allows the user to make as many requests as he desires and check for their completion anytime later.

Looking deeper into this idea, the gateway can generate a universally unique identifier for the request and use it as a key for a new entry in a database. By sending this identifier to both the worker in the message and the user in the reply, the former will update the table with the result after finishing, while the gateway will be able to retrieve it whenever the latter asks for the output.

However, the results' table is not the only necessary, as the message queue should not be a database itself, that is, it should not store the data sets to be used, since the messages ought to be as small as possible, guaranteeing that the queue's performance is not degraded. Doing so also has a positive impact on scalability, given that the message queue will be able to handle many more messages, leaving the data storage to a component optimised to do so. Furthermore, this indicates that both the gateway and the worker may communicate with the data service. A data service that contains multiple tables is therefore needed, resulting in a new architecture represented in Figure 5.4.

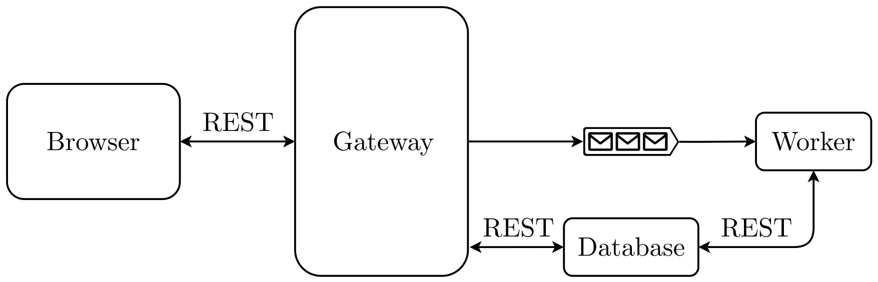


Figure 5.4: Server architecture with the data service.

This new architecture comes at a price, and adding this data service could raise some privacy concerns since Chapter 3 classifies security as of the utmost importance. Therefore, either should the databases follow Section 2.2.6 or the entire back-end should be implemented in the servers of the centre where the original data is already stored, being accessible only to those within the organisation. As for authorisation, a JWT authentication mechanism should be implemented, preventing users other than the owner from having access to the data sets and results. The resulting token must then be sent in the `Authorization` header of every HTTP request. The architecture will then evolve into Figure 5.5.

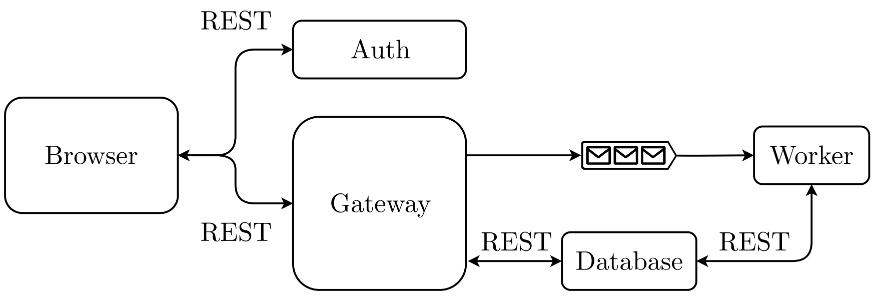


Figure 5.5: Architecture with authentication service.

Finally, a simple email service to notify the users of their tasks' completion, as well as a logging service – essential for microservice architectures – that centralises every service's logs, should be added to the system, resulting in the diagram presented in Figure 5.6. Since logs do not require any reply, this service reads every record asynchronously from a message queue, while the latter receives REST requests, to ensure emails are successfully sent.

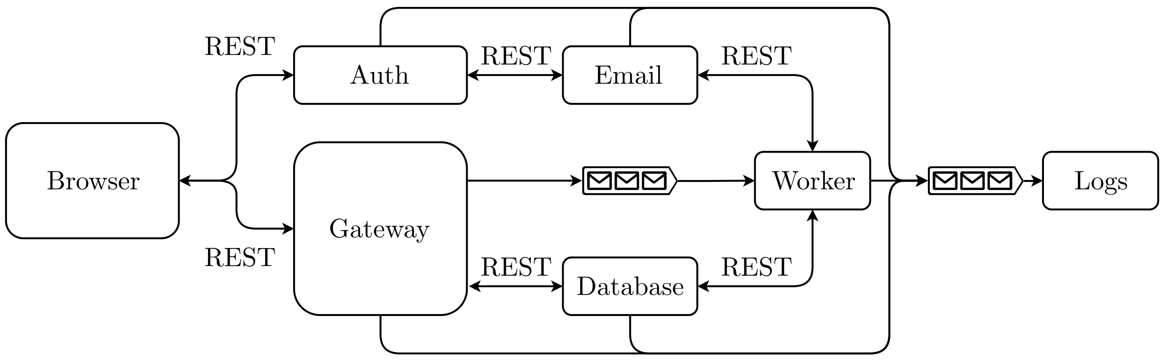


Figure 5.6: Final architecture.

5.5 Worker architecture

As already discussed, the workers retrieve tasks from a message queue to execute asynchronously. These messages should have a structure similar to the one in Figure 5.7, which is only an example and will vary according to the requested package but always including the following keys:

- **address** *(string)* – Email address to be used when sending the result, if **notification** is `true`;
- **inputFiles** *(dict)* – Contains every file to be used by the package, where the keys are the files' names as detailed in the package's interface (explained below). Since the `learnDBM` package only takes a `-i` file, the given example indicates that this should be the one identified by `datasetid`;
- **link** *(string)* – URL for the results' webpage. If **notification** is `true`, the worker will concatenate this URL with the **requestId** and send it to the given **address** via the email service;

```
{
  "address": "useremail",
  "inputFiles": {
    "-i": "datasetid"
  },
  "link": "resultlink",
  "method": "learnDBM",
  "notification": true,
  "params": {
    "-ind": 1,
    "-k": 2,
    "-m": 1,
    "-ns": false,
    "-p": 1,
    "-sp": false,
    "-tDBN": true
  },
  "requestId": "requestid",
  "requestName": "requestname",
  "time": "2020-10-13T15:01:29.721657",
  "userId": "userid"
}
```

Figure 5.7: Message received by the worker when running the `learnDBM` package to find two clusters in the `combinedDataset.csv`, a stationary tDBN structure, one intra-slice in-degree, a Markov lag of one and one parent.

- **method** *(string)* – A string that corresponds to the package to be executed;
- **notification** *(Boolean)* – Whether the user wants to receive an email notification when the result is available and if the execution takes longer than 30 seconds;
- **params** *(dict)* – A dictionary with the package's parameters and their values, represented as key-value pairs, respectively. If the values are Boolean, the parameters will simply be set;
- **requestId** *(string)* – The id of this request, used to place the result in the database;
- **requestName** *(string)* – Request's name, given by the user, which will be used to name the output and resulting files, if any;
- **time** *(string)* – Coordinated universal time in ISO format. This will be used to determine whether 30 seconds have passed since the request was placed if **notification** is `true`;
- **userId** *(string)* – The user's id.

After polling and retrieving the message, the worker proceeds to import the files from the data service and build the command to execute by parsing the message's **params**. Here, it should be noted that the packages are executed as terminal commands, as to allow any type of program to run, independently of their development language, improving the system's extensibility – if a user can run it in using the command line, so should this tool. To build the command, the worker makes use of the package's interface, which should be presented in a JSON file with the method's name. Figure 5.8 exemplifies this file.

Once again, while the contents will differ from package to package, the interface should always consist in a JSON object with the following fields:

```
{
  "cmd": [
    "java",
    "-jar",
    "packages/learnDBM.jar",
    "-mt",
    "-pm",
    "-o",
    "clusters.csv",
    "-d"
  ],
  "params": {
    "-ns": false,
    "-sp": false,
    "-bcDBN": false,
    "-cDBN": false,
    "-ind": 0,
    "-k": -1,
    "-m": 1,
    "-p": 1
  },
  "inputFiles": {
    "-i": "Input CSV file to be used for network learning"
  },
  "outputFiles": {
    "dbn.txt": {},
    "dbn.json": {},
    "clusters.csv": {}
  }
}
```

Figure 5.8: Interface for the `learnDBM` package – `learnDBM.json`.

- **cmd** *(strings list)* – The base command to execute the package as a list of strings, to which the remaining parameters will be added. For this case, if the user were to execute the package locally, the command would start with `java -jar packages/learnDBM.jar -mt -pm -o clusters.csv -d`, assuming the executable was saved in a directory called "packages";
- **params** *(dict)* – A dictionary containing the flags accepted by the package, as key-value pairs, used to filter the request. When the value is a Boolean, the flag takes no value, and should simply be either present or absent. Otherwise, the value is either the default value or a representation of the type of input (whether it is a number, a string, etc.), but will not affect building the command. Ergo, this interface does not check for mandatory parameters, leaving it to the package to throw the correct error and message, which will be used as output;
- **inputFiles** *(dict)* – A key-value representation of the accepted input files, where the value is a description of the file used for the corresponding key;
- **outputFiles** *(dict)* – The only field unrelated to building the command is a dictionary containing as its keys the files that the package will output, which should be saved in the database. As for the values, while they are empty dictionaries in the running example, this only happens when the file has no metadata. When it does have this extra data, the dictionary will have a **table** key, containing

the name of the table that should hold the file's metadata, followed by this information represented as key-value pairs and, finally, it might even have an **original** field, reserved for whenever the result inherits metadata from one of the **inputFiles**. To accommodate these unknowns, this **original** attribute consists of key-value pairs where the keys must also be present in **inputFiles** and the values are dictionaries consisting of the following attributes:

– **table** *(string)* – The table containing the original file's metadata;

– **attributes** *(string list)* – A list with the metadata attributes to copy from the original file.

Figure 5.9 represents a more complex example of this field, taken from the `eqw` package's interface. This package exposes the need for this field since, when discretising a file, it is impossible to know whether it will contain missing values – this depends on the original file having them or not, since the discretisation methods do not impute any values.

```
{
  "output.csv": {
    "table": "datasets",
    "discrete": true,
    "original": {
      "-i": {
        "table": "datasets",
        "attributes": [
          "missing"
        ]
      }
    }
  }
}
```

Figure 5.9: **outputFiles** field for the `eqw` package's interface.

Since the resulting command is built with user input and ran as a terminal instruction, it is of utmost importance to not only use an extensible yet secure interface that filters out invalid input but also call the command without invoking a shell, to prevent against command injection exploitation [70]. Otherwise, a user could potentially make a request where `-i` is equal to `; rm -rf /`, generating the `java -jar packages/learnDBM.jar -mt -pm -o clusters.csv -d -i ; rm -rf /` command, which, after running the package, deletes the worker's entire root directory. However, the mentioned measures secure against these attacks, as the first item in the command will be the only program to run, while the rest are its arguments. In this way, the command becomes `java`, which gets executed with the arguments `["-jar", "packages/learnDBM.jar", "-mt", "-pm", "-o", "clusters.csv", "-d", "-i", "; rm -rf /"]`, resulting in a controlled error that is of no threat to the worker, as it is caught and flagged as such. The same happens when the packages themselves halt, with the worker uploading the fault message to the database.

It now becomes clear that there is no need to have each package as a microservice: the extensibility is achieved by allowing packages to be easily added/removed by simply including/deleting the JSON interface and the executable, which can even be done without stopping and rebuilding this component.

5.6 Data service

While the data sets and other files should be stored in tables to allow for easy operations such as lookups, their size is not compatible with this solution, as data tables usually have a maximum size per item. However, they cannot be simply stored as files in a file system either, as not only does every file have metadata related to it (for instance, whether a data set has missing values), but some results might also not even have a corresponding file. Therefore, a solution may be to mix both options and use a data table to store the metadata and a file system for the files. This results in some endpoints returning files' metadata and ids instead of the files themselves, as their contents should be fetched separately.

Even though the file system is a straightforward solution, the same cannot be said about the data table, as either a relational or a NoSQL one can be elected. Taking Section 2.2.5 into account, and considering that having a BASE database is acceptable, as this is not a real-time application, and that there will be no need for complex queries, a NoSQL system should be preferred due to its edge on scalability, improved performance, overall ease of use, and flexibility of the stored data structures.

Having chosen the technology to be used, while the files are stored in a separate folder for each user, there should be four distinct data tables:

- **datasets** – This should contain the uploaded data sets' metadata, and its id should match a file in the file system;
- **networks** – Here, the ids for files with DBN models should be stored, as well as the id of the request that created the network;
- **others** – A table that is in everything similar to the **datasets**, but with files that are not intended to be available as TS for analysis;
- **results** – Where the results, as well as information on their output and input files, are saved.

Here, it might be argued that the **datasets** and **others** tables could be merged by using an extra parameter which differentiated them, since items in NoSQL tables do not follow specific schemas and, besides the additional metadata, these tables are similar. As true as this is when considering two tables, so is the fact that this merger does not scale as well, with the query performance taking a hit when filtering the results based on this extra key, since there is a distinct endpoint for each table and no case where a list of both will be retrieved. This reasoning applies just as well to the **networks** and **results**.

Additionally, note that any package may upload files to tables other than the **results** one, since the output files can be data sets and networks themselves, meaning that they will not only appear as output files but also constitute an entry in the respective table.

Finally, this service's endpoints, used by the gateway and the workers, are listed below. Since they can only be accessed within a private network, there are no authentication issues to address.

`/examples`

Where the gateway `GET`s a list of the example data sets, receiving a JSON with an array of dictionaries containing the files' names and intended methods.

`/examples/{name}`

This endpoint allows to `GET` a specific example CSV file.

`/methods`

To `GET` a list of dictionaries with each available method's name and characteristics of its main input file, which are equivalent to the **mainFile** field of the `/methods/{method}` `GET` response.

`/methods/{method}`

The gateway may `GET` a `method`'s template, to know its parameters and input files that allow to dynamically build the front-end, through this endpoint. This template, represented in Figure 5.10, should not be confused with the package's interface in the worker service, having the following fields:

- **checkboxes** *(dict)* – Every Boolean parameter as key-value pairs where the key is the parameter's name and the value is its description;
- **fields** *(dict)* – Key-value pairs representing text or number input fields where the keys are the parameters' names and the values are dictionaries containing their descriptions and default values;
- **files** *(dict)* – Every input file (except the main one) represented as key-value pairs, where the key is the file's input name and the value is its description;
- **mainFile** *(dict)* – A dictionary containing the name of the main input file and its metadata (for instance, if it must be discrete or have missing values);
- **method** *(string)* – The method's name;
- **options** *(dict)* – Key-value pairs with the parameters that have multiple options, where the keys are their names and the values have their descriptions and a list of the possible values.

`/{userid}`

To `DELETE` an entire user's data (objects and files), the gateway must use this endpoint.

`/{userid}/{fileid}`

Knowing in advance the `fileid` (as the worker may deduce it from the request's id), this endpoint allows to `POST` and `GET` a specific file to/from the file system.

`/{userid}/{table}`

When the gateway `POST`s to this endpoint, a new entry is added to the `table` and the resulting id is returned. It is also possible to `GET` a list of names and ids of the entries in the specified user's table.

`/{userid}/{table}/{fileid}`

This endpoint is used to `GET`, update (`PUT`) or `DELETE` entries and files, identified by `fileid`.

```
{
  "checkboxes": {
    "-ns": "Learn non-stationary network",
    "-sp": "Force intra-slice connectivity to be a tree"
  },
  "fields": {
    "-ind": {
      "description": "Intra-slice in-degree",
      "value": 1
    },
    "-k": {
      "description": "Number of clusters",
      "value": 2
    },
    "-m": {
      "description": "Markov lag",
      "value": 1
    },
    "-p": {
      "description": "Number of parents",
      "value": 1
    }
  },
  "files": {},
  "mainFile": {
    "name": "-i",
    "metadata": {
      "discrete": true,
      "missing": false
    }
  },
  "method": "learnDBM",
  "options": {
    "type": {
      "description": "DBN structure",
      "values": [
        "-tDBN",
        "-bcDBN",
        "-cDBN"
      ]
    }
  }
}
```

Figure 5.10: Template for the `learnDBM` method.

5.7 Gateway API endpoints

Before implementing the back-end, it is necessary to document the API endpoints, that is, the URLs that will be used to access the data, make requests, and any other relevant interaction with the services. These may be split into three classes, as they can be either authenticated and unauthenticated calls, or even requests where the user's authentication is irrelevant. Note that when not specified, the back-end might return `5XX` errors if an internal error occurs and return `200` on success.

5.7.1 Unauthenticated requests

These calls should only be done by unauthenticated users, as they relate to registering an account and logging the user in, to receive a JWT.

`/auth/confirm`

After signing up, the user should receive an email with a confirmation code, so that every address is verified. By `POST`ing a JSON object with the code and the email address, the back-end checks if the code matches and replies with a `200` code if it is correct.

`/auth/login`

To log into the website, the user must `POST` his email address and password to this endpoint. If the request is successful, he will then receive a JWT back. If the address has already signed up but is yet to verify the email, a new code is sent.

`/auth/refresh`

Whenever the user's access token expires, he may use this endpoint to `GET` a new one. While this requires the `Authorization` header, this should be the refresh token, so this request is not considered authenticated.

`/auth/signup`

This endpoint allows to `POST` a user's email address and password to sign himself up. If the email is not already registered, a message is sent to the address with a confirmation code, to be later used in the `/auth/confirm` endpoint.

5.7.2 Authenticated requests

It is after logging in that the users have access to most of the endpoints. For these, the `Authorization` header should carry the access JWT, so that the gateway can not only verify if the user is allowed to perform these actions, but also decode his id. Thus, this last information is never directly required by any endpoint.

`/datasets`

This endpoint allows to `GET` the user's data sets as an array of names and ids. The user will also `POST` to this endpoint whenever he wants to upload a new data set, sending the details and file in the request body. As a reply, he will get the new data set's id.

`/datasets/{datasetid}`

To `GET` a specific data set's information as a JSON response, the user should indicate the id in the URL. It is also possible to `DELETE` the data set entirely from the back-end.

/files/{fileid}

Using this endpoint, the client can GET the content of a file, given its id, or DELETE it if needed.

/methods/{method}

Whenever the user wants to analyse a data set, he should POST to this endpoint, using the method's name in the URL and passing the parameters to the method in the body. As a response, he will receive the id for the future result. Figure 5.11 represents an exemplifies the body of these requests, whose fields are explained below:

```
{
  "inputFiles": {
    "-i": "datasetid"
  },
  "params": {
    "-ind": 1,
    "-k": 2,
    "-m": 1,
    "-ns": false,
    "-sp": false,
    "-p": 1,
    "-tDBN": true
  },
  "requestName": "requestname",
  "notification": true
}
```

Figure 5.11: Body of a request using the `learnDBM` package with a stationary tDBN structure, one intra-slice in-degree, a Markov lag of one, two clusters and one parent.

- **inputFiles** *(dict)* – Contains every file to be used by the package, where the keys are the files' names as detailed by the package's interface. Since the `learnDBM` package only takes a `-i` file, the given example indicates that this should be the one identified by `datasetid`;

- **notification** *(Boolean)* – Whether the user wants to receive an email notification when the result is available and if the execution takes longer than 30 seconds;

- **params** *(dict)* – A dictionary with the parameters of the packages and their values, represented as key-value pairs, respectively. If the values are Boolean, the parameters will simply be set;

- **requestName** *(string)* – Request's name, given by the user, which will be used to name the output entry and files, if any.

/networks

As one of the methods allows to create DBNs, which can be used to make predictions, it is necessary to have an endpoint to GET a list of the networks' names and ids.

`/network/{networkid}`

This endpoint allows to `GET` and `DELETE` a network. Since the network's id is the same as the id of its original request, this endpoint essentially redirects to the `/results/{resultid}` endpoint.

`/others`

This is similar to the `/datasets` endpoint, but to `GET` and `POST` files that are not data sets, such as files for inference.

`/others/{otherid}`

The "other" files can be downloaded with a `GET` request or deleted with a `DELETE` HTTP call.

`/results`

By making a `GET` request to this endpoint, the user will retrieve his results' ids and names as an array.

`/results/{resultid}`

After getting the id of a new request, the user can use this endpoint to `GET` or `DELETE` the result. If the package is still processing the request, the call will wait up to 30 seconds for it to finish. If that time is not enough, it will reply with a `504` error ("Endpoint request timed out"). However, if the given id does not exist (either because it never existed or because it has already been cleared from the database), a `404` error ("Not found") is produced. On success, it returns a `200` with the result in the body of the response. This body may vary from package to package, but Figure 5.12 is an illustration of a possible clustering response.

Even though the result varies according to the package, data set and parameters, the main structure is common across every request, with the keys having the following meanings:

- **done** *(Boolean)* – The state of the request. Should be always `true`, as the request will only return a `200` code when the package has finished its execution;
- **errors** *(Boolean)* – Specifies if there were any errors in the worker, specifically when running the package;
- **files** *(dict)* – Contains every file generated by the worker. In this dictionary, the keys are the files' names as outputted by the package and the values are dictionaries containing the files' names and ids, so that the user can request them from the back-end individually;
- **inputFiles** *(dict)* – Contains every file that was used by the package, where the keys are the files' names as detailed by the package's interface. Since the `learnDBM` package only takes a `-i` file, the given example indicates that this should be the one identified by `datasetid`;
- **method** *(string)* – The method requested, which produced this result;

```json
{
  "done": true,
  "errors": false,
  "files": {
    "clusters.csv": {
      "datasetId": "requestid_clusters.csv",
      "datasetName": "requestname_clusters.csv"
    },
    "dbn.json": {
      "datasetId": "requestid_dbn.json",
      "datasetName": "requestname_dbn.json"
    },
    "dbn.txt": {
      "datasetId": "requestid_dbn.txt",
      "datasetName": "requestname_dbn.txt"
    }
  },
  "inputFiles": {
    "-i": "datasetid"
  },
  "method": "learnDBM",
  "output": {
    "bic": -116800.9730292124,
    "clusters": 2,
    "logs": [
      "Number of networks with max score: 1",
      "Network score: -81.3409407547621",
      "Number of networks with max score: 1",
      "Network score: -71.04462891866356",
      "Starting with stochastic EM."
    ],
    "observations": 2000
  },
  "pending": false,
  "requestId": "requestid",
  "requestName": "requestname",
  "userId": "userid"
}
```

Figure 5.12: Response obtained after requesting the `learnDBM` package using the `combinedDataset.csv`, a stationary tDBN structure, one intra-slice in-degree, a Markov lag of one, two clusters and one parent.

- **output** *(JSON)* – The package's standard output will be parsed as a JSON object, so that it can be easily read by the consumer. In case of an error, the standard error output will be listed here, usually as a list of strings;
- **pending** *(Boolean)* – Whether the result is still waiting for the **files** to be inserted. The back-end should only return a `200` response when this entry is `false`;
- **requestId** *(string)* – The identifier of the requested result, which should correspond, in the end-point, to the `resultid`;
- **requestName** *(string)* – Name of the result, given by the user;
- **userId** *(string)* – The user's id.

`/users/me`

The user can also `DELETE` his account using this endpoint, consequently deleting all his files and results.

5.7.3 Requests with or without authentication

The requests concerning logging out and retrieving mehods' templates or example data sets to test the packages can be done by both authenticated and unauthenticated users, using the following endpoints.

`/auth/logout`

The user may `POST` his refresh JWT to this endpoint so that it is invalidated and can no longer be used to generate access tokens. This call does not require the `Authorization` header, as the refresh token will be revoked regardless. Hence, this is not considered an authenticated request and can be done by anyone.

`/examples`

Only allows a `GET` request, to retrieve the list of example data sets available, as an array of names and intended methods.

`/examples/{exampleid}`

Endpoint to `GET` an example data set as text, in CSV format.

`/methods`

The user may `GET` a list of dictionaries with each available method's name and the characteristics of its main input file using this endpoint.

`/methods/{method}`

The user may `GET` a specific `method`'s template using this endpoint. This request is then redirected to the `/methods/{method}` endpoint of the data service, detailed in Section 5.6.

5.8 Emails and logs

Both the logging and the email services are rather simple: the former reads messages solely containing logs to write to a file, while the latter receives `POST` requests on its `/` endpoint with bodies composed by the `subject`, `message` and a list of recipient `mails`.

Chapter 6

Implementation

As discussed in Chapter 2, the best way to guarantee this web tool's longevity and scalability is by implementing it in the cloud using a serverless stack, if possible. However, medical and academic institutions might need to deploy the application on-premises either to ensure their data does not leave a managed network or to include their own packages. In this chapter, we analyse both of these back-end implementations, which are accessed by identical Angular client applications.

6.1 Cloud

Since there is no one-size-fits-all service when it comes to the cloud, this section focuses on breaking down an optimal architecture that is robust enough to withstand pressing workloads while being consistently suitable for sporadic demand, starting with evaluating multiple cloud providers and choosing one based on the so-far mentioned requirements and constraints.

The resulting cloud implementation – Figure 6.1 – is accessible through API Gateway [71] via REST requests and Cognito [72] authentication, which uses JWTs itself. Both Lambda [73] and Elastic Compute Cloud (EC2) [74] workers are used for processing, with the latter being dynamically launched per task queued using Amazon Simple Queue Service (SQS) [75]. The workers' logic is similar and both retrieve data from DynamoDB [76] and Simple Storage Service (S3) [77], sending notifications through Amazon Simple Email Service (SES) [78]. Moreover, AWS Identity and Access Management (IAM) [79] is used for privacy assurance when accessing S3 and Amazon CloudWatch [80] allows monitoring each service.

As for the front-end, the chosen platform to deploy the Angular application that makes use of this back-end was GitHub Pages due to its longevity, scalability and no-cost availability.

6.1.1 Choosing a cloud provider

First and foremost, price is of the essence when conceptualising an architecture or choosing a cloud provider. With an unlimited budget, the latter would hardly matter, and the former could be effortlessly over-provisioned. However, for this project to remain feasible, costs must be kept to a minimum, starting with choosing where to host the back-end.

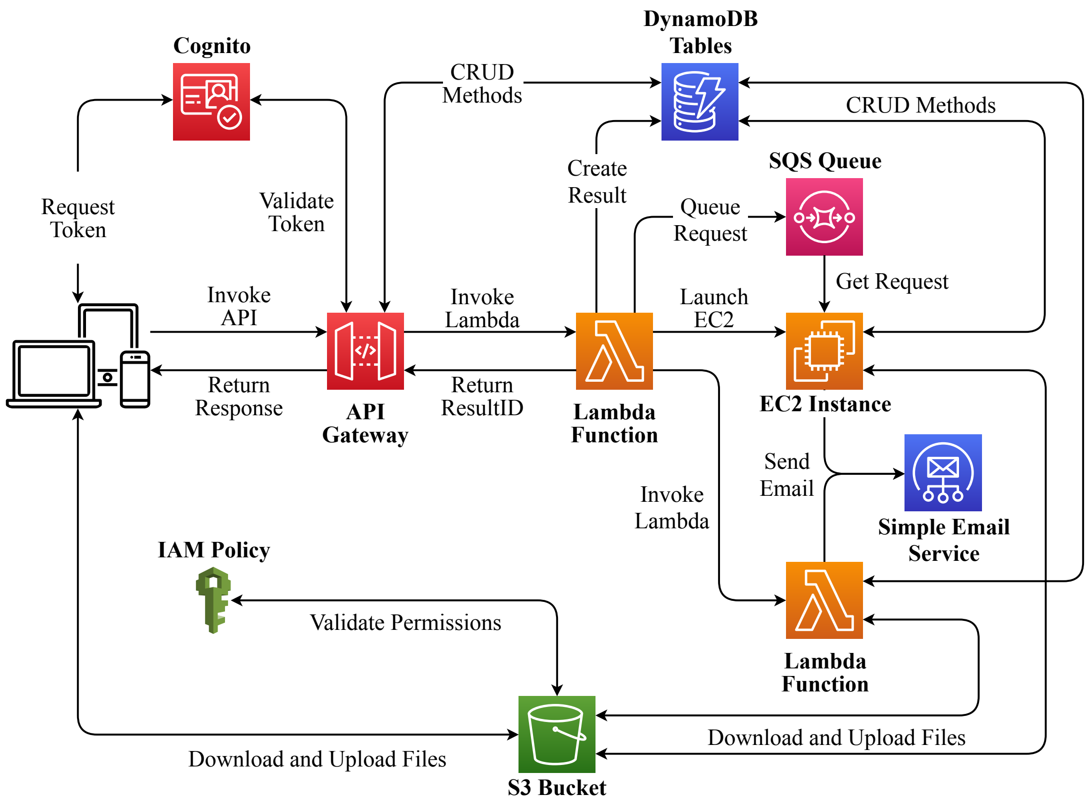


Figure 6.1: Complete cloud architecture.

While the world is currently facing a rise in cloud providers [81], we will focus on four of the major solutions [82]:

- Amazon Web Services (AWS) [83];
- Google Cloud [84];
- Microsoft Azure [85];
- Heroku [86].

Firstly, Heroku is the only to exclusively offer one type of service – PaaS –, which runs on top of Amazon's IaaS [87]. Moreover, it does not adopt a pay-as-you-go policy, meaning the customer pays in advance for a specific tier, which is not ideal for sporadic traffic. Not only that, but the free tier has no encryption solutions and does not scale well, since it has only one worker, which would need to execute tasks in parallel threads. For these reasons, this platform can be immediately ruled out.

On the other hand, the remaining three providers implement equivalent services between themselves, with consequently identical pricing plans. Since all encompass the needed tools and are capable of satisfying the minimum requirements, we focused on investigating which had the lowest and most consistent average cold start latency for both VMs [88, 89] and serverless functions [90] while continually providing excellent performance [91]. For these reasons, and since it is a pioneer in cloud computing that has endured through time, we elected AWS, whose main building blocks are its IaaS and FaaS services: EC2 and Lambda, respectively.

6.1.2 Developing the workers

Perhaps the most relevant component in the back-end, the worker must be able to scale well with the number of requests, support any programming language, and be readily available to execute pending tasks. Not surprisingly, this is also where cost and the clients' sporadic demand are most pertinent.

The immediate solution to ensure high availability would be to have a server listening to the queue and executing the packages as soon as tasks are published. Doing so leads to the scenarios in Figure 2.7, which can be improved using a load balancer and an auto-scaling group that automatically scales up and down the number of servers, ensuring that there is always the right capacity while reducing the time where the demand exceeds it. This time can even be null if new servers are spawned before the available ones reach their maximum capacity. However, a trade-off between unnecessary costs and computational power arises: the better the servers, the higher the idle charge will be. Since the requests are expected to be relatively infrequent, albeit arriving in large volume, having at least one server always running and waiting for tasks implies that it will be idle most of the time. Moreover, if each server is sized to handle multiple exponentially complex requests, its fixed cost will necessarily be significant.

To mitigate this fixed cost while ensuring scalability with every request, an EC2 instance can be launched whenever a client needs to analyse a data set. Since API Gateway cannot do this on its own, a Lambda function should connect the two components. As such, the former will redirect the request to the latter, which will then queue the task using Amazon SQS and launch an EC2 VM running a Docker [92] container. This process is schematised in Figure 6.2 and can be seen as an auto-scaling that has the advantage of not needing a server manager nor to pay for unnecessary preemptive servers.

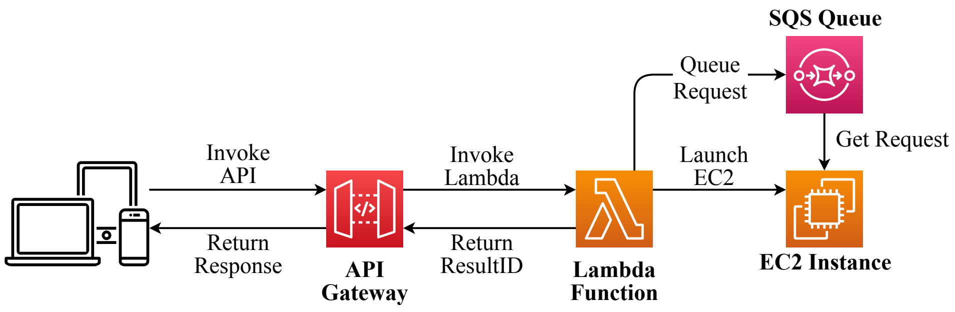


Figure 6.2: Cloud architecture that scales with the number of requests.

Notwithstanding the above, cold start, defined as the time it takes to have the VM running after requesting it, is a major problem with the proposed approach. Even if some studies have proved that Amazon has the overall best startup times which are continuously decreasing, the fact is that they still go beyond 15 seconds (and may even take up to a few minutes) and depend on the deployment location as well as the time of the day [88, 89]. Since it is not acceptable for a user to wait for minutes while requesting one-second tasks, the so-far designed architecture is still insufficient. However, going back to Section 2.2.7, a serverless architecture would allow to instantly scale up as much as needed with low latency [90]. Therefore, using Amazon's answer to FaaS – AWS Lambda –, it is possible to execute the tasks in parallel without having any cold start, resulting in the architecture depicted in Figure 6.3.

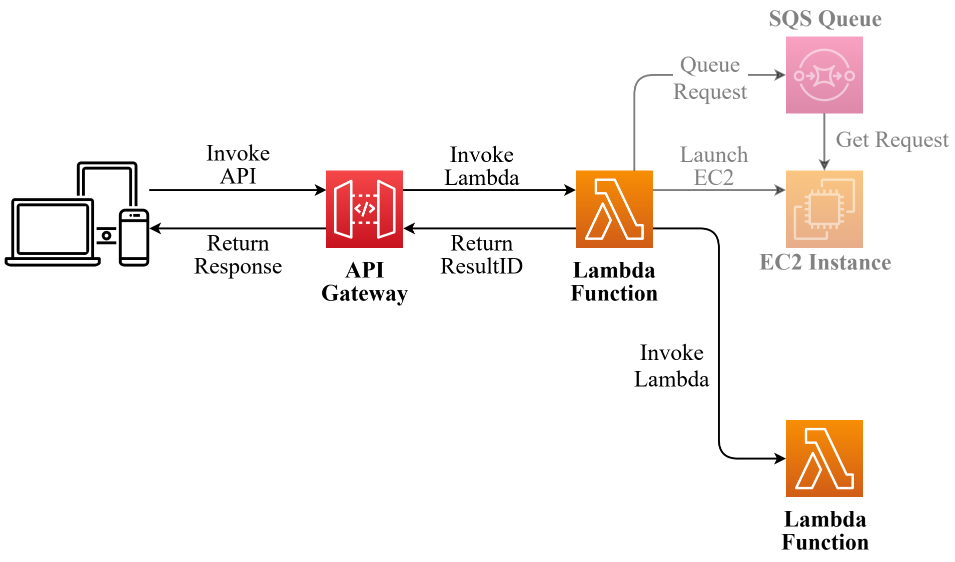


Figure 6.3: Cloud architecture that immediately scales with the number of requests.

But why not resorting solely to FaaS? Although these scale as much as necessary and without noticeable cold starts, there are still some limitations: they are subject to execution time limits (15 minutes for AWS), have much lower computational power (being comparable only to the most basic VM instance types) and do not accept every programming language out of the box. Thus they are best suited to fill the time taken to launch EC2 instances and complete those requests that take a few minutes or less, with the guarantee that their failures do not compromise the tasks' completion.

Finally, serverless containers are also a noteworthy concept for this section, as they try to merge Lambda's scalability and lack of server maintenance with EC2's extensibility in terms of allowing to run a Docker container with almost no limits. Some notable cases are AWS Fargate [93] and Google Cloud Run [94], but while the former still has the same cold start issues – essentially just abstracting away the servers themselves –, the former is only now starting to allow tasks to run for up to one hour.

6.1.3 Adding the data service

Since the workers cannot perform their tasks without input files and taking into account that users must be able to upload files of any size, this service must follow the architecture described in Section 5.6 while ensuring complete availability.

To implement the already conceptualised service, both a NoSQL database and a file system were needed. For Amazon, these are DynamoDB and S3 respectively, and both can be accessed by the workers using AWS software development kit (SDK). The same is not true for the users: even though they upload and download S3 files directly (resorting to the AWS Amplify [95] framework for Angular), the data tables are accessed using API Gateway. Ideally, both would be available through the API, but this is not possible since Amazon limits the requests' payload size to 10 MB. This results in the back-end portrayed in Figure 6.4, which does not have the `/files` or the data service's endpoints.

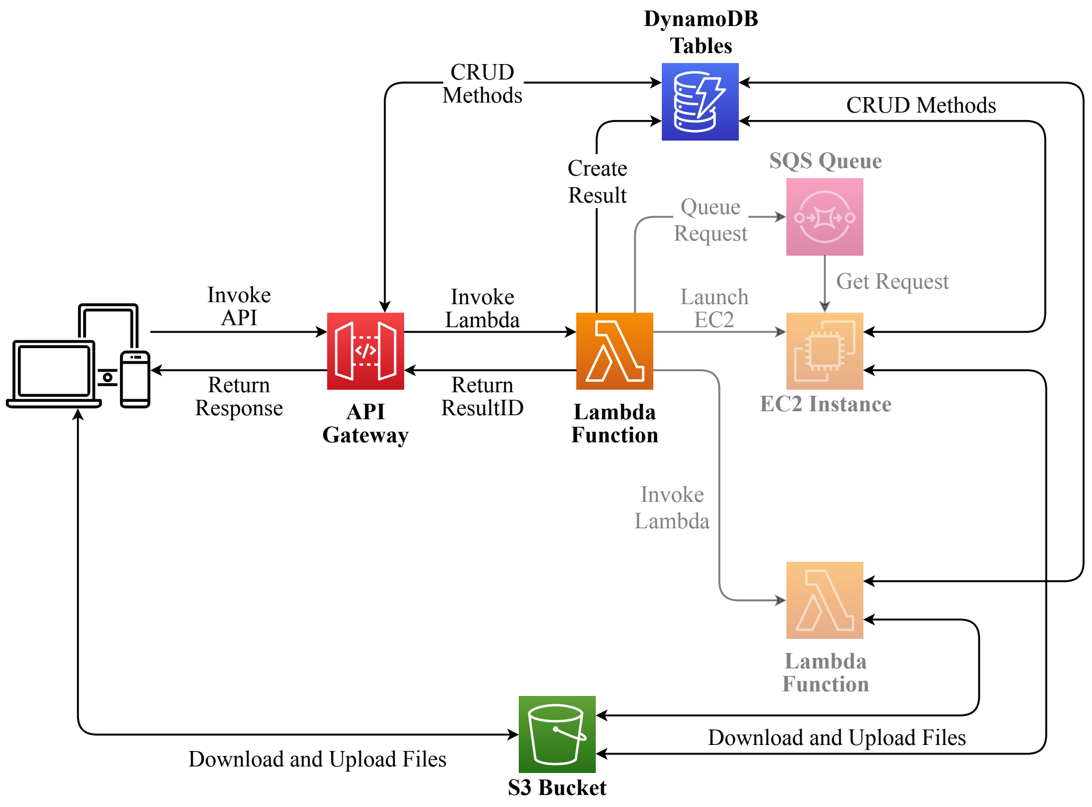


Figure 6.4: Cloud architecture including the data service.

6.1.4 Ensuring privacy

Since the users retrieve files directly from S3 and must make authenticated requests to API Gateway, Amazon's authentication service – Cognito – is also needed. This component takes care of managing the User Pool, generating JWTs and controlling service access.

Cognito essentially replaces the `/auth/` endpoints detailed in Section 5.7.1, as these authentication actions are also included in AWS Amplify. API Gateway then confirms the JWT's validity and retrieves the user. Moreover, when the application tries to access S3, the storage service will check whether the Cognito User has the necessary permissions resorting to AWS IAM. These interactions result in Figure 6.5.

6.1.5 Alerting the users

There is also no need to develop an email service since AWS has its own: Amazon SES. When any of the workers finish running the packages or encounter an error, they invoke this service using Amazon's SDK. On the other hand, Cognito does not use SES, as it sends address confirmation emails itself. Figure 6.6 represents the cloud architecture after adding this component.

6.1.6 Keeping track of what is happening

Instead of creating a logging service, Amazon CloudWatch may be used, since it allows monitoring every components' usage and gathering their logs. This piece of the architecture is not represented in the implementation scheme since it is inherent to every AWS service.

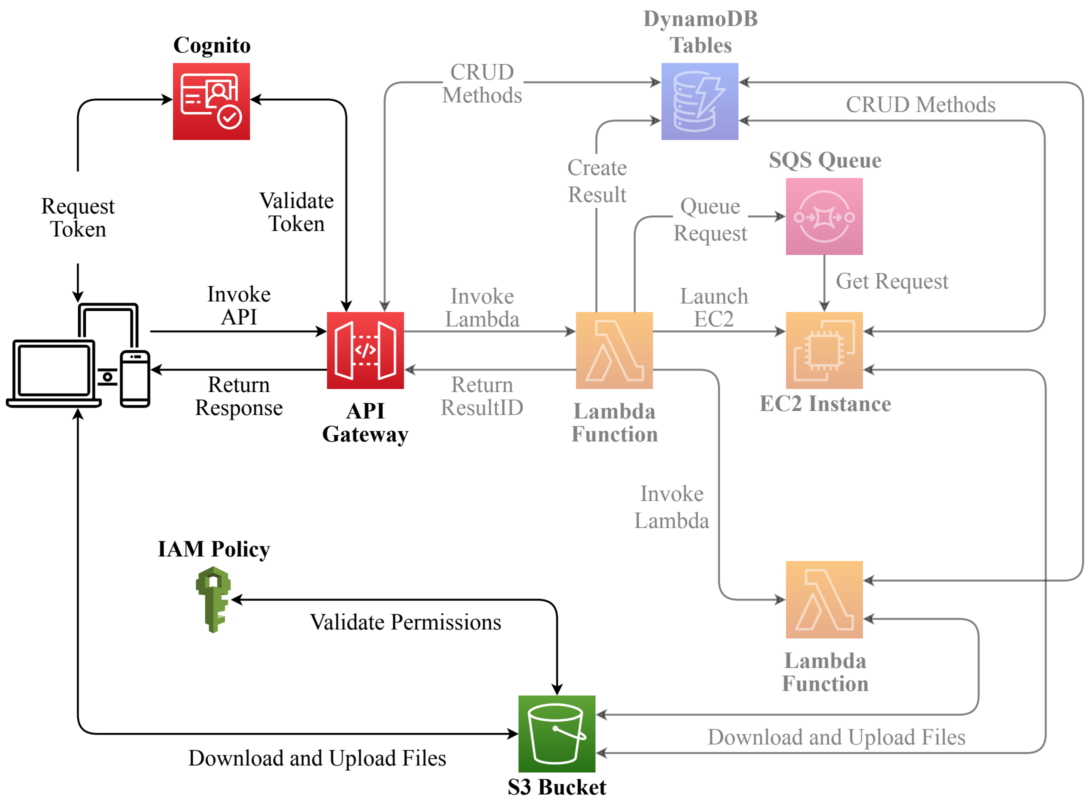


Figure 6.5: Cloud architecture including user authentication.

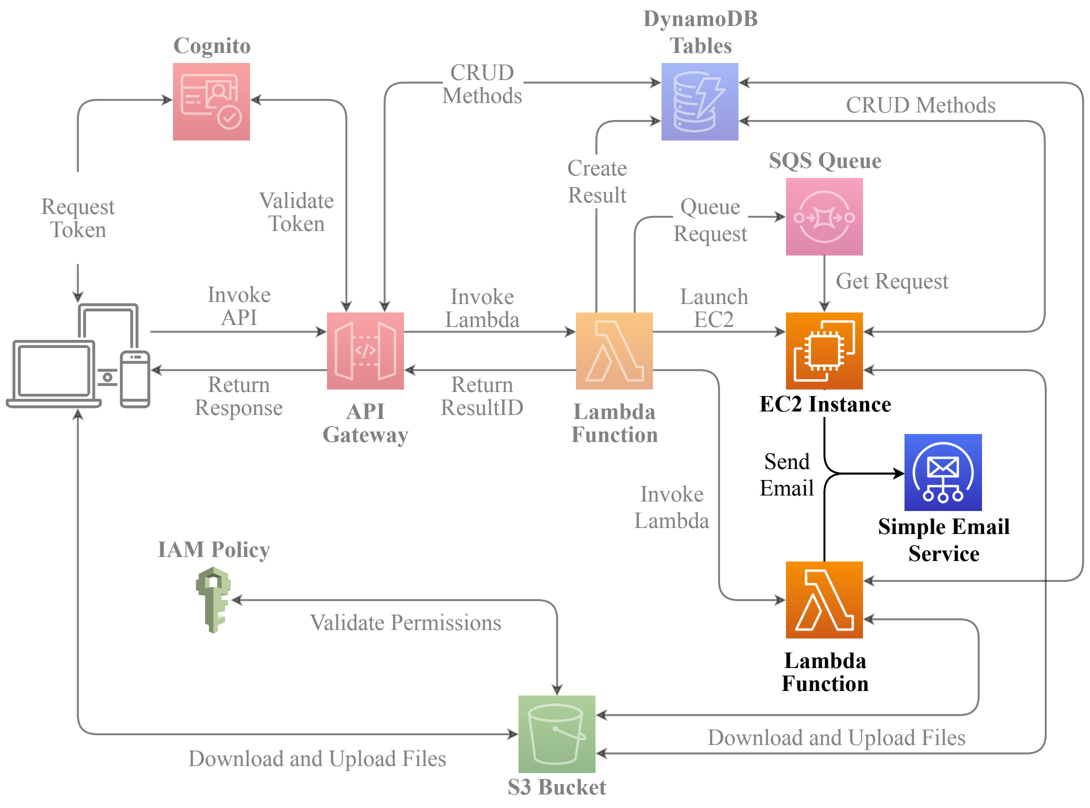


Figure 6.6: Cloud architecture that allows sending emails.

6.2 On-premises

For the intranet version of the application, the back-end can be developed using Docker containers which are orchestrated using Docker Compose [96], facilitating its deployment and management. This approach follows the advantages of using containers, detailed in Section 2.2.7, and leads to the network depicted in Figure 6.7, with the gateway and auth services being merged due to their simplicity and similarities.

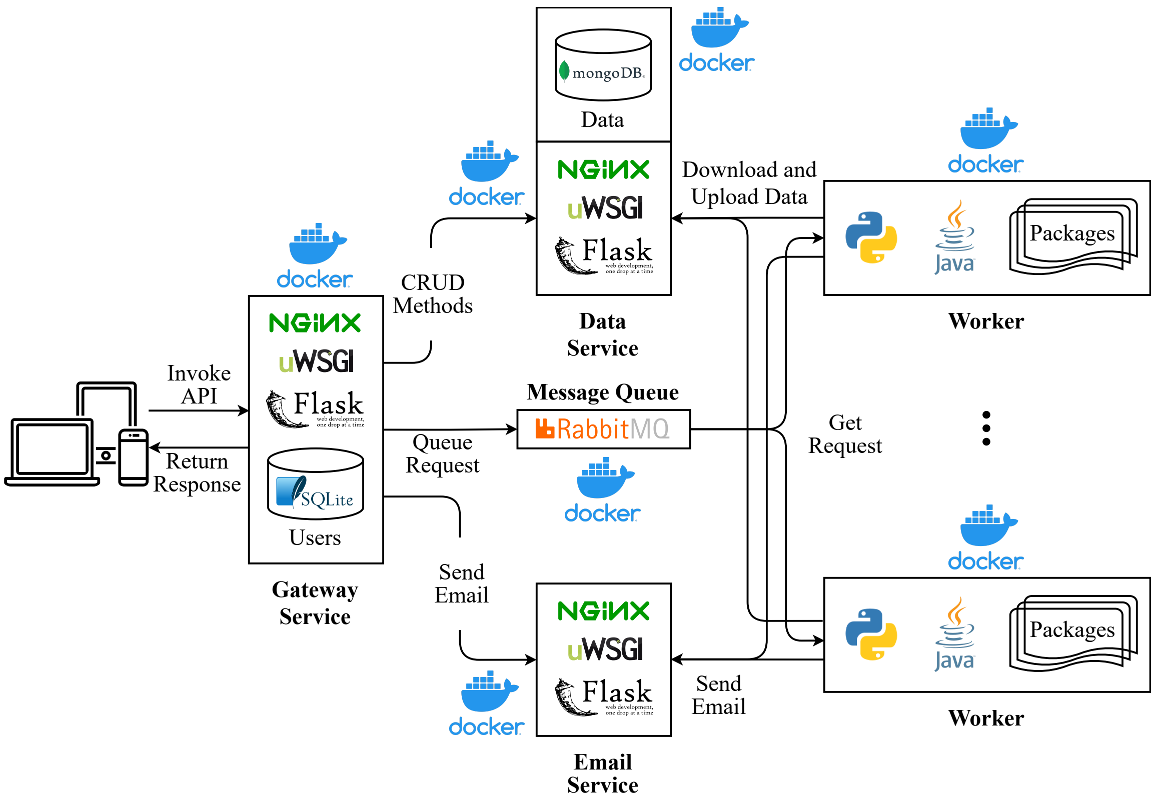


Figure 6.7: On-premises architecture.

To implement the services, the traditional NGINX, uWSGI and Flask architecture [97, 98] was chosen, with all of them running on an Alpine Linux container, where NGINX is the web server and reverse proxy, uWSGI is the application server, and Flask is the application itself, where the logic written in Python lies. As for the gateway, NGINX also serves the Angular static files, to be fetched when the user first accesses the website, and its Flask program uses an SQLite [99] table to save the authentication information.

In this architecture, the number of workers no longer corresponds to the number of requests, but to the number of servers running these containers, which can be easily added to the network. This is not to say they can only handle one request at a time, as they launch a new thread for each package execution. And while, by default, the workers come with Java and Python installed, as the former is needed to run most of the packages and the latter to fetch messages from a RabbitMQ [100] work queue and execute the analyses, any other language can be included by adjusting this service's Dockerfile [101]. In addition, if there are no extra servers to add more workers when they exhibit signs of resource starvation, it is always possible to limit the number of messages each worker can process simultaneously, allowing them to process pending tasks before retrieving any more from the queue.

Moreover, for the database tables, the predominant NoSQL MongoDB [102] management system was adopted, due to its ease of use and remarkable performance [103]. As for the logs, similarly to the cloud implementation, a logging service was not designed, since Docker handles this centralised role itself. When running containers as part of Docker Compose, every service's logs are displayed on the screen and saved to a dedicated file.

Finally, although Docker Compose will run all the containers in the same server by default, they were developed to be distributed over a network. This means that it is possible not only to spin up as many containers as necessary in different hosts but also to have each service deployed in an adequate server (for instance, the data service needs a lot of free disk space and fast I/O operations whereas the email only needs minimal resources). Furthermore, this orchestration tool guarantees that services are restarted if they halt for any reason.

Chapter 7

Results

Now that MAESTRO – Figure 7.1 – has been designed and implemented [7], it is time to put it to test. This chapter begins with a demonstration of the web tool's potential by going through how a researcher could use it to analyse a real data set, which is followed by performance assessments on the cloud implementation and a discussion on which requirements were met and how each of them is delivered.

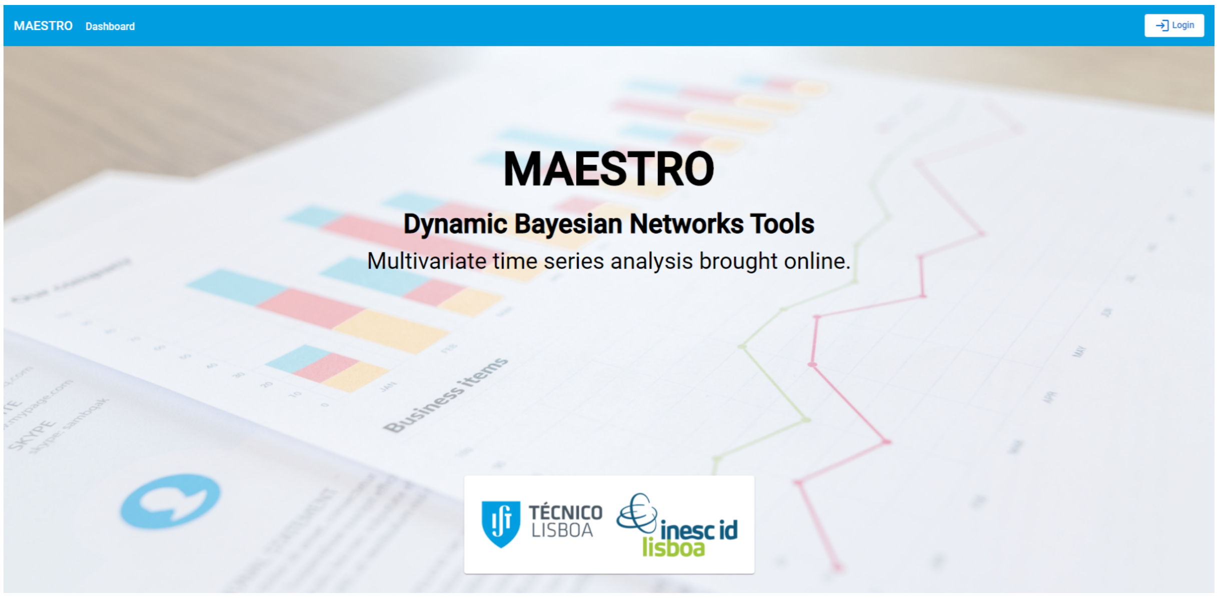

MAESTRO
Dashboard
Login
MAESTRO
Dynamic Bayesian Networks Tools
Multivariate time series analysis brought online.
TÉCNICO LISBOA
inesc id lisboa

Figure 7.1: MAESTRO's landing page.

7.1 Case study

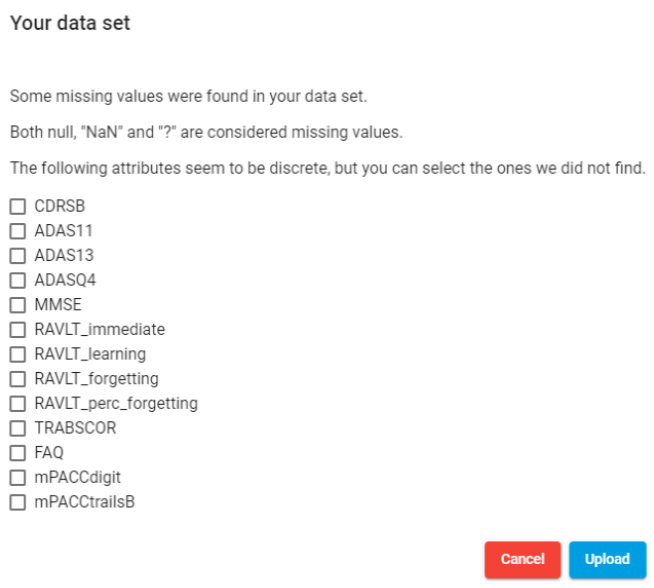
The web tool's usage is now illustrated with real data extracted from the Alzheimer's Disease Neuroimaging Initiative (ADNI) database [104], which has been extensively analysed in various studies [105, 106]. The ADNIMERGE data set consists of 13,413 time series with 113 attributes (including identifiers, dates and other codes) measured from 1,973 patients through multiple case report forms – such as Alzheimer's Disease Assessment Scale (ADAS) or Rey's Auditory Verbal Learning Test (RAVLT) – and biomarker lab summaries across the ADNI protocols.

Patients that had examinations after 24 months were selected, as suggested in previous studies [107]. Afterwards, we merged duplicates and inserted empty rows for the missing observations, allowing us to compute the percentage of missing values per feature. By removing every attribute where these exceeded 20 %, we picked the 13 features presented in Table 7.1, which reduced the number of patients to 1,286.

Table 7.1: Continuous features selected from ADNIMERGE and their ranges, averages (A), medians (Mdn) and percentages of missing values (%m).

Feature	Range	A	Mdn	%m
`CDRSB` – Clinical Dementia Rating Scale sum of boxes	$[0, 17]$	1.8	1	16.8
`ADAS11` – ADAS (11 items)	$[0, 70]$	10.6	9	16.5
`ADAS13` – ADAS (13 items)	$[0, 71.3]$	16.5	14.3	17.3
`ADASQ4` – ADAS with delayed word recall	$[0, 10]$	5.1	5	16.5
`MMSE` – Mini-Mental State Examination	$[1, 30]$	27.0	28	16.4
`RAVLT_immediate` – Sum of five RAVLT trials	$[0, 75]$	35.0	34	16.8
`RAVLT_learning` – RAVLT trial 5 minus trial 1	$[-5, 14]$	4.1	4	16.8
`RAVLT_forgetting` – RAVLT trial 5 minus delayed recall	$[-9, 15]$	4.3	4	17.0
`RAVLT_perc_forgetting` – RAVLT_forgetting divided by trial 5	$[-450, 100]$	58.9	60	17.5
`TRABSCOR` – Trail Making Test part B	$[0, 996]$	119.5	89	18.7
`FAQ` – Functional Activities Questionnaire	$[0, 30]$	4.7	1	16.6
`mPACCtrailsB` – Modified Preclinical Alzheimer's Cognitive Composite (mPACC) with TRABSCOR	$[-39.7, 12.9]$	-5.5	-3.7	16.4
`mPACCdigit` – mPACC with Digit Symbol Substitution	$[-39.7, 7.1]$	-5.7	-4.1	16.4

When uploading the data – Figure 7.2 –, the application detects missing values and assumes attributes are continuous (albeit the user may specify otherwise).


Your data set

Some missing values were found in your data set.

Both null, "NaN" and "?" are considered missing values.

The following attributes seem to be discrete, but you can select the ones we did not find.

- ☐ CDRSB
- ☐ ADAS11
- ☐ ADAS13
- ☐ ADASQ4
- ☐ MMSE
- ☐ RAVLT_immediate
- ☐ RAVLT_learning
- ☐ RAVLT_forgetting
- ☐ RAVLT_perc_forgetting
- ☐ TRABSCOR
- ☐ FAQ
- ☐ mPACCdigit
- ☐ mPACCtrailsB

Figure 7.2: Result of selecting the file to be used. MAESTRO automatically interprets missing values and which attributes are discrete, but the user may select more.

The visualisation of the Mini-Mental State Examination (MMSE) is shown in Figure 7.3 and allows us to see some patterns. In this case, it is clear that most patients do not have examinations for the fourth time point and usually tend towards high (blue) values. For visualisation purposes, continuous values are binned by equal intervals, but hovering over them will reveal their real value. To observe another attribute, there is a drop-down menu where the user can select any feature.

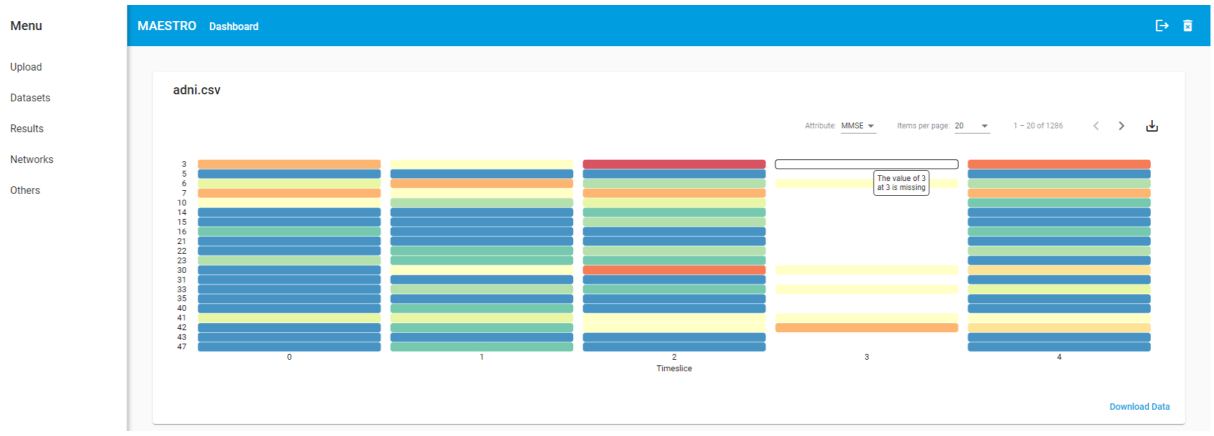


Figure 7.3: Visualising the patients' MMSE results before imputation and discretisation.

Since this data is continuous and has missing values, we are given the option of either imputing or discretising the time series, with the first one being restricted to LOCF and LR. Using the latter results in the data presented in Figure 7.4. As expected, the missing values were filled using linear interpolation.

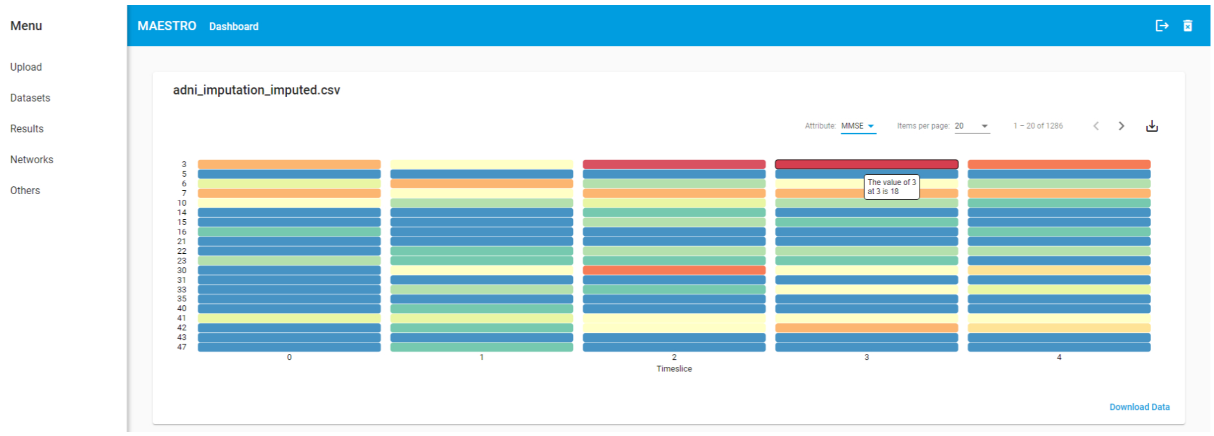


Figure 7.4: Result of using linear regression to impute the data.

Subsequently, we discretise the data using EQF and three bins (representing low, medium and high values). For MMSE, the following bins are generated:

- a – $[1, 27]$;
- b – $[28, 29]$;
- c – 30.

Doing so produces the data represented in Figure 7.5.

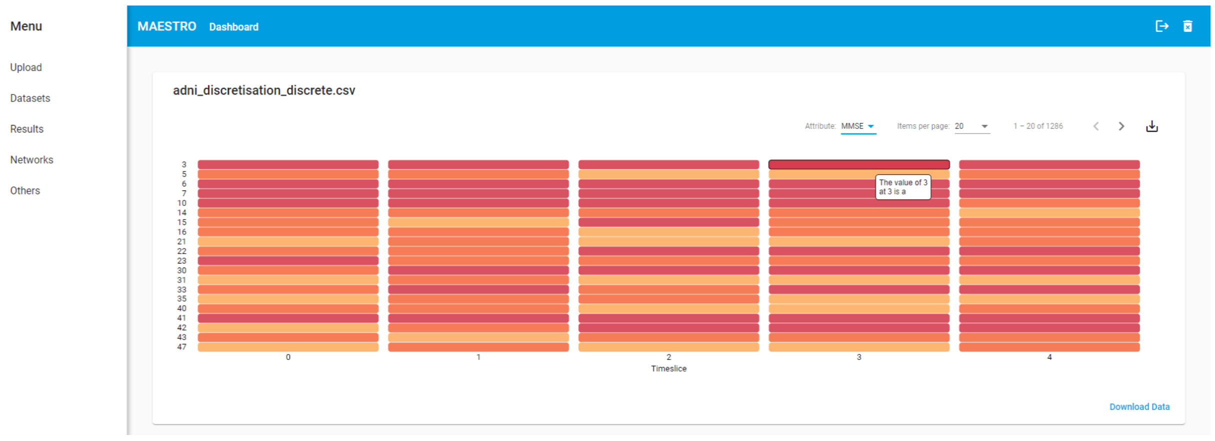


Figure 7.5: Result of discretising the data using EQF and three bins.

Following that, we proceed to clean the data by removing outliers. Through learning a tDBN using log-likelihood as the scoring criterion, a Markov lag of one and two parents, it is possible to plot the outlierness of each transition and subject in a histogram – Figure 7.6.

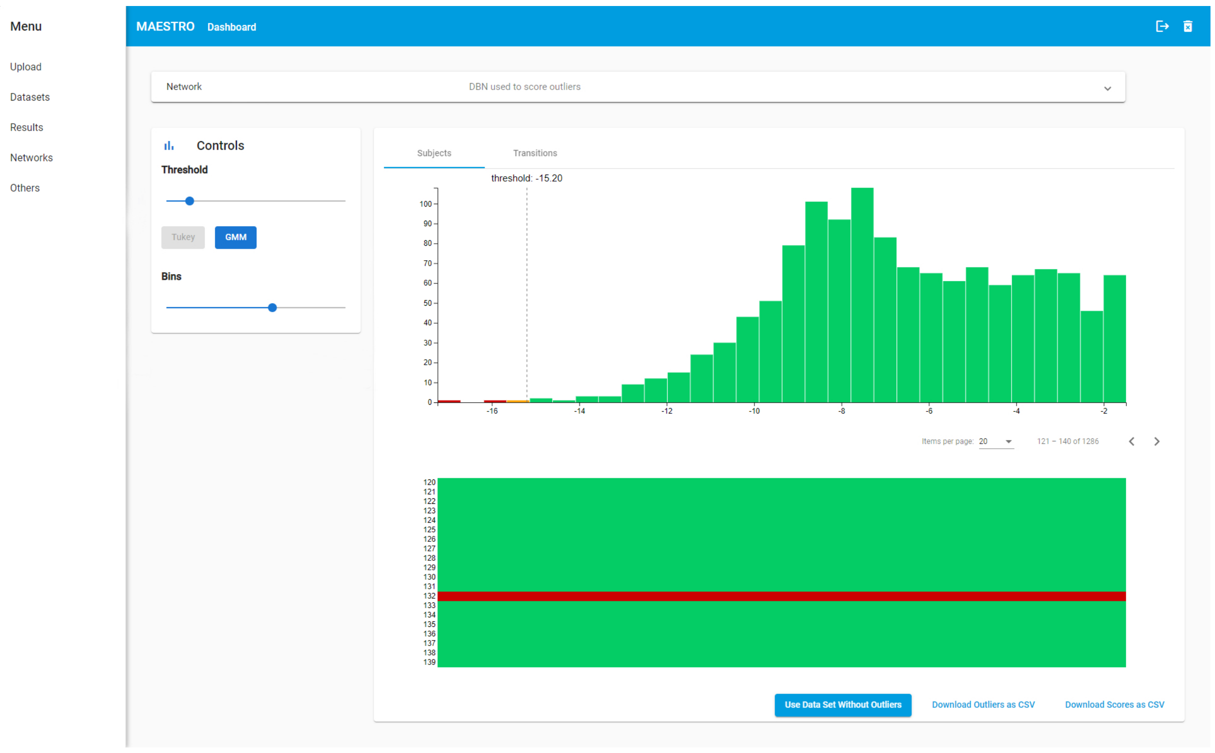


Figure 7.6: Result of outlier detection using log-likelihood as the scoring criterion, a Markov lag of one and two parents. The threshold is set to the result of using Tukey's strategy.

In this step, the user may tweak the number of bins in the histogram and change the score not only to Tukey's and GMM's results but also to any other value. Through the histogram, we have an idea of how the data is getting split, with the lower panel showing exactly which subjects or transitions are being classified as outliers. Moreover, the user may expand the upper card and reveal the resulting network, which can be then downloaded as either a text file or an image. For this case, the representation of the network is shown in Figure 7.7.

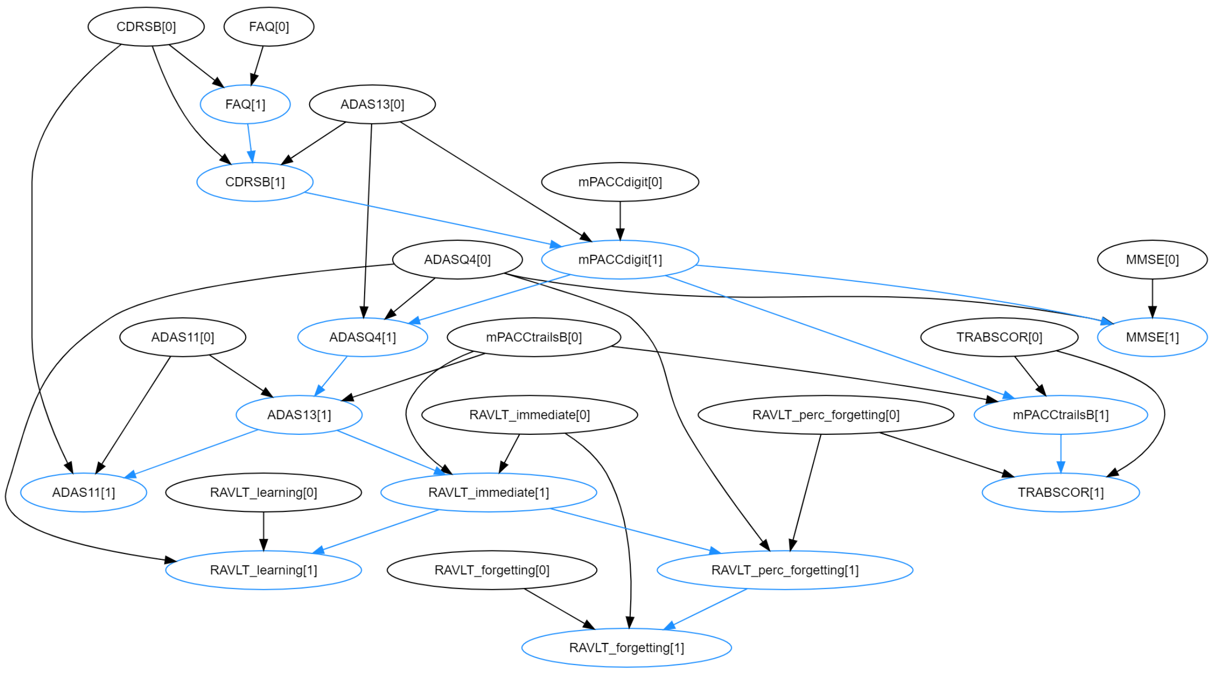


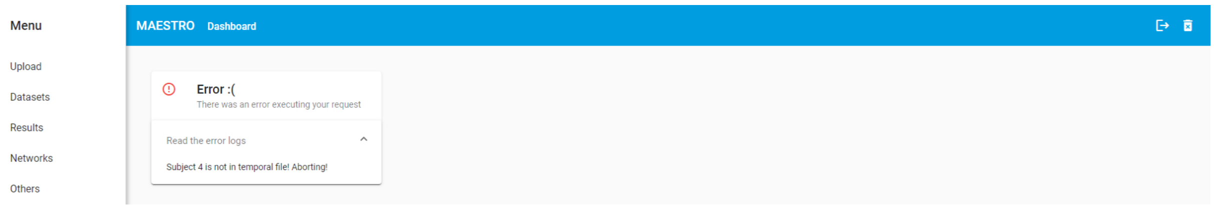
Figure 7.7: Network learnt for outlier detection using log-likelihood as the scoring criterion, a Markov lag of one and two parents.

Having removed the three outlier subjects using Tukey's threshold, we now analyse the remaining patients, starting with training an sdtDBN model for inference. Since this method accepts static attributes, we selected the three variables described in Table 7.2.

Table 7.2: Static features from ADNIMERGE, their classes and percentages of missing values (%m).

Feature	Classes	%m
`PTGENDER` – Gender	{Male, Female}	0.0
`PTETHCAT` – Race	{Not Hisp/Latino, Hisp/Latino}	0.5
`PTMARRY` – Initial marital status	{Married, Divorced, Widowed, Never married}	0.3

Adding these observations, we may train the sdtDBN using a log-likelihood scoring criterion, a unitary Markov lag, one dynamic parent, two static ones, and no restrictions. However, knowing this package cannot process static files with subjects that do not exist in the dynamic input, we added a new one to induce an error. As seen in Figure 7.8, this is gracefully caught and presented to the user.



Error :(

There was an error executing your request

Subject 4 is not in temporal file! Aborting!

Figure 7.8: Error handling in the application.

When using the original observations, the result is successful. Again, this network may be viewed through an expandable card, where it is also possible to download it – Figure 7.9.

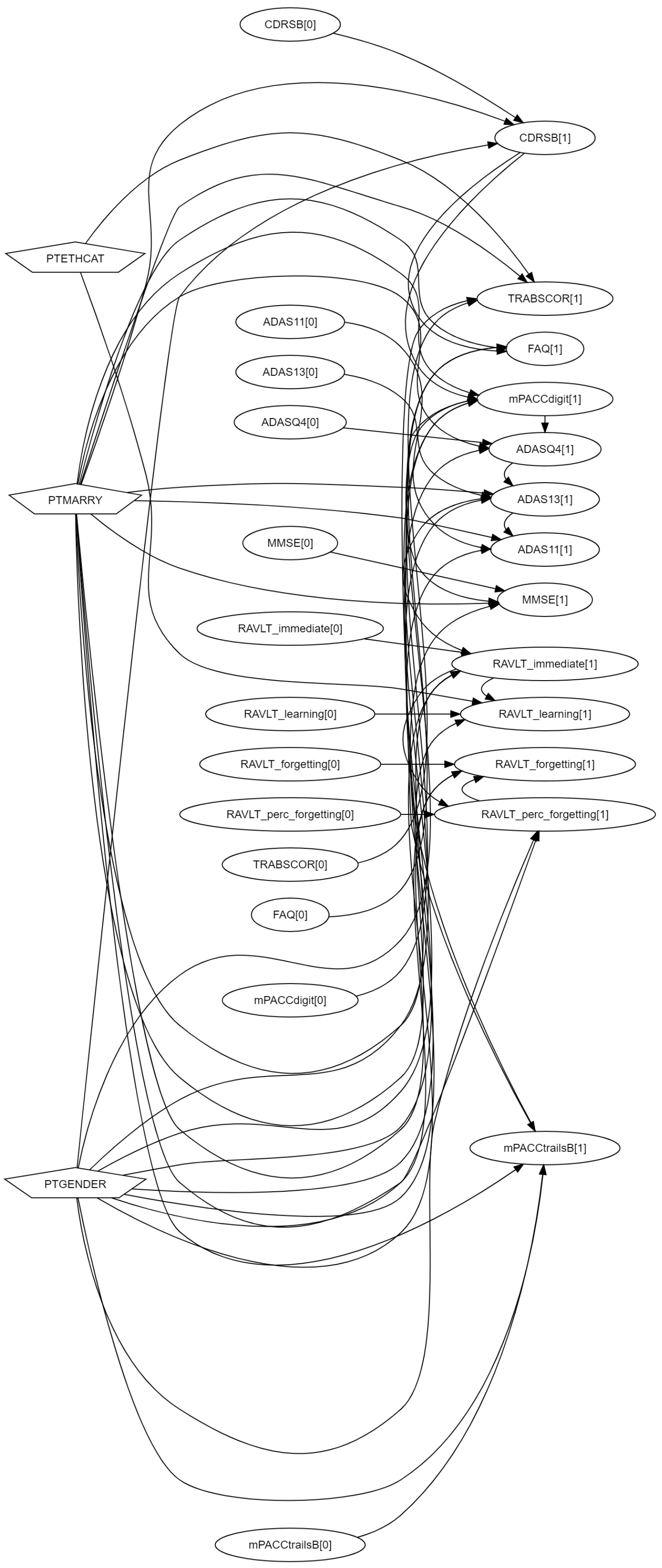
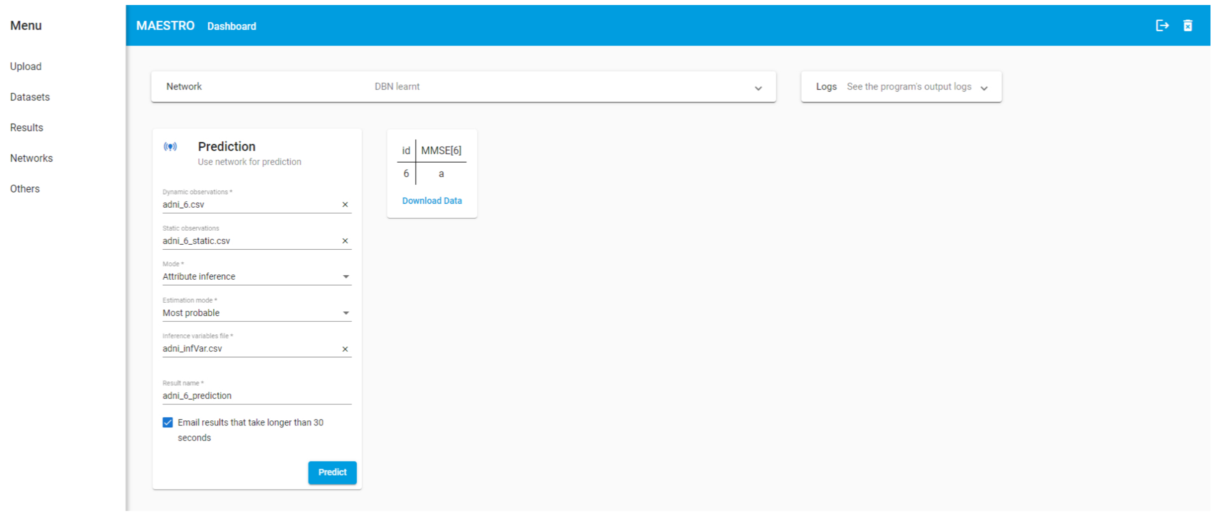


Figure 7.9: DBN learnt using log-likelihood as the scoring criterion, a unitary Markov lag, one dynamic parent and two static ones.

Knowing the MMSE value for subject 6 at the seventh time point is 22 (falling into bin `a` according to this discretisation), we can test the model's prediction by feeding it this subject's dynamic and static observations and requesting the most probable inference of this attribute at the 36th month. The model makes an accurate prediction, as can be seen in Figure 7.10. Moreover, even though the prediction was solely made on one subject, attribute and time point for testing purposes, the request may include any number of the aforementioned, which are added to the table.



Network: DBN learnt

Logs: See the program's output logs

Prediction
Use network for prediction

Dynamic observations *: adni_6.csv

Static observations: adni_6_static.csv

Mode *: Attribute inference

Estimation mode *: Most probable

Inference variables file *: adni_infVar.csv

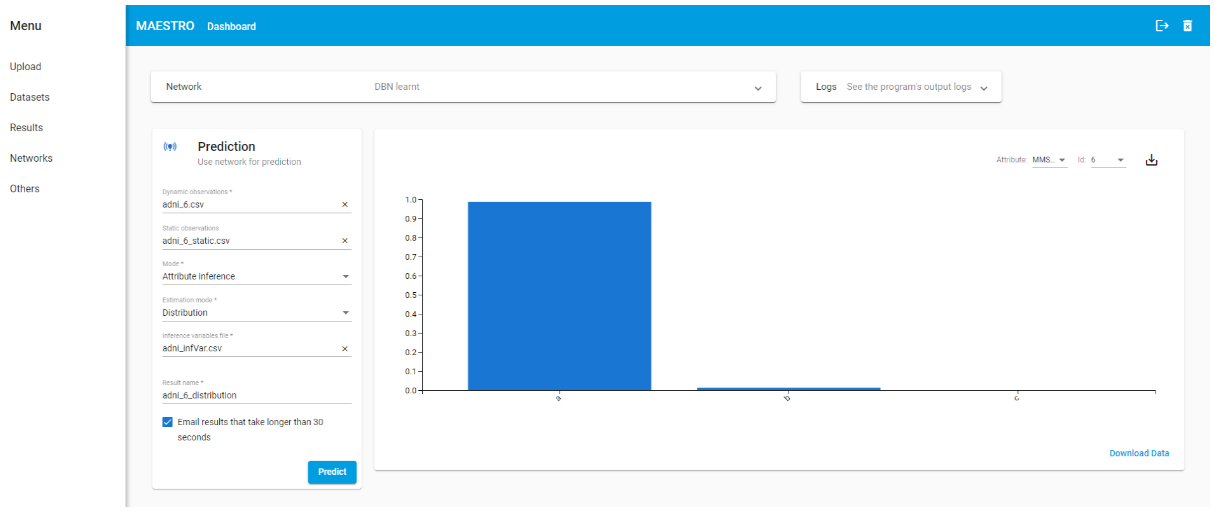
Result name *: adni_6_prediction

☑ Email results that take longer than 30 seconds

id	MMSE[6]
6	a

Figure 7.10: Most probable MMSE for subject 6 at the seventh time point.

To further understand this result, it is possible to use the same dashboard to compute this attribute's distribution of probabilities for the requested subject and time point. Upon completion, the new result is a histogram that replaces the previous table and graphically represents the probability of each class. From Figure 7.11, it is clear that the likelihood heavily inclines towards the predicted class. However, there is some fluctuation on these probabilities across multiple requests, since the intermediate attributes are predicted using random sampling according to each node's probability distribution. There is also a drop-down for when the request contains multiple analyses.



Network: DBN learnt

Logs: See the program's output logs

Prediction
Use network for prediction

Dynamic observations *: adni_6.csv

Static observations: adni_6_static.csv

Mode *: Attribute inference

Estimation mode *: Distribution

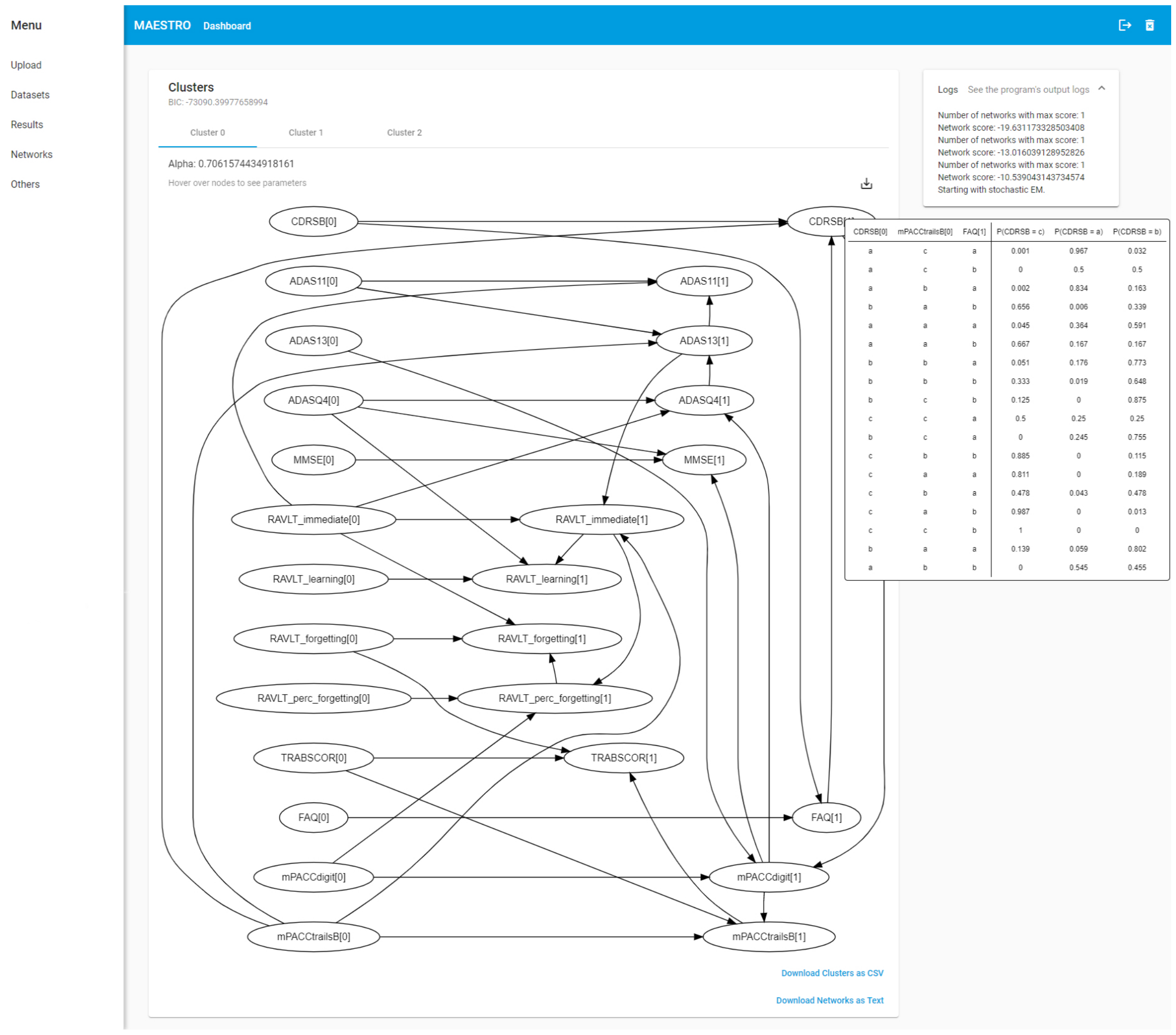
Inference variables file *: adni_infVar.csv

Result name *: adni_6_distribution

☑ Email results that take longer than 30 seconds

Figure 7.11: MMSE's predicted probability distribution for subject 6 at the seventh time point.

Finally, we may group the TS into clusters. Since there are three possible diagnoses – clinically normal, mild cognitive impairment, and dementia –, we try to fit three tDBNs using an intra-slice in-degree of one, two parents and a unitary Markov lag. After handling the request, MAESTRO presents the results as in Figure 7.12. It is also possible to download both the network as text and the scores or clustering results as CSV files. Furthermore, every network representation of each cluster is presented in an individual tab.



Clusters

BIC: -73090.39977658994

Alpha: 0.7061574434918161

Logs See the program's output logs

```
Number of networks with max score: 1
Network score: -19.631173328503408
Number of networks with max score: 1
Network score: -13.016039128952826
Number of networks with max score: 1
Network score: -10.539043143734574
Starting with stochastic EM.
```

CDRSB[0]	mPACCtrailsB[0]	FAQ[1]	P(CDRSB = c)	P(CDRSB = a)	P(CDRSB = b)
a	c	a	0.001	0.967	0.032
a	c	b	0	0.5	0.5
a	b	a	0.002	0.834	0.163
b	a	b	0.656	0.006	0.339
a	a	a	0.045	0.364	0.591
a	a	b	0.667	0.167	0.167
b	b	a	0.051	0.176	0.773
b	b	b	0.333	0.019	0.648
b	c	b	0.125	0	0.875
c	c	a	0.5	0.25	0.25
b	c	a	0	0.245	0.755
c	b	b	0.885	0	0.115
c	a	a	0.811	0	0.189
c	b	a	0.478	0.043	0.478
c	a	b	0.987	0	0.013
c	c	b	1	0	0
b	a	a	0.139	0.059	0.802
a	b	b	0	0.545	0.455

Figure 7.12: Result of finding three clusters using tDBNs with a Markov lag of one, two parents and an intra-slice in-degree of one.

Notice how, as in every other network displayed by the application, hovering over a node will show a table with the conditional probabilities that describe it. Moreover, as in most of the other DBN-related functionalities, it is possible to view the application's logs, which correspond to the information that the command-line executable usually prints to the standard output.

This might lead to the assumption that errors are only caught when the packages quit successfully since the workers are retrieving the standard output. As such, it should be clarified that, even though the error shown in this section is a controlled one, any exception that causes the packages to quit unexpectedly is also caught. However, these may be too verbose if not handled properly by the program.

7.2 Stress testing

7.2.1 Vertical scalability

One of the fundamental requirements for this website was ensuring it was capable of handling every long-running analysis within an acceptable time frame, which is tightly related to the system's vertical scalability: if we are able to augment the processing nodes' computational resources, the execution time will decrease. To assess this metric in the cloud architecture, we chose the `learnDBM` package for its higher resource demand and used the `combinedDataset.csv` introduced in page 31 with a Markov lag of two, a unitary intra-slice in-degree, two parents, two clusters and multithreading. The test was conducted by launching three distinct EC2 instance types – m5.large, m5.xlarge and m5.2xlarge –, connecting to each of them, and executing a Python script which sequentially started and timed ten child processes that performed the aforementioned task. Since the purpose was to evaluate the hardware impact on execution time, we opted not to include any network-bound operations in the measurements and solely timed the package execution. The results are plotted in Figure 7.13.

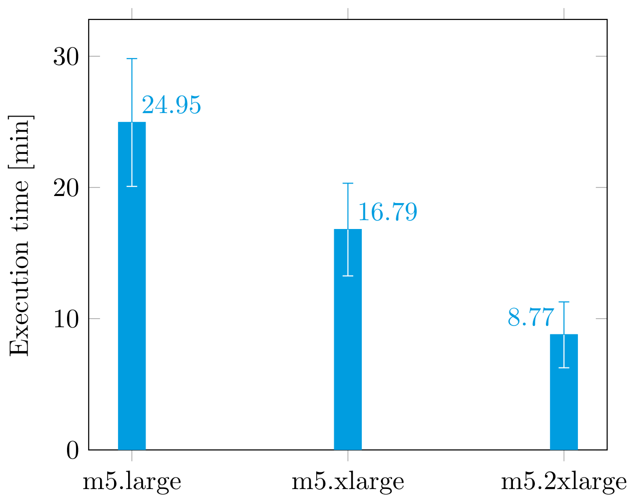


Figure 7.13: Execution time when clustering the `combinedDataset.csv` for ten times in three different instances with a Markov lag of two, a unitary intra-slice in-degree, two parents, two clusters and multithreading.

As expected, the most powerful VM – the m5.2xlarge with 8 virtual processors and 32 gibibytes of memory – was substantially faster than the one with the least resources – the m5.large with 2 virtual processors and 8 gibibytes of memory. The vertical scalability is thus warranted, since the developer may easily choose to specify a more robust instance to be used for subsequent requests whenever these are taking too long.

Furthermore, it is worth mentioning that using extra computing resources may not be significantly more expensive, since the instances are terminated after the requests complete. For instance, even if the m5.xlarge – with 4 virtual processors and 16 gibibytes of memory – costs $ 0.222 per hour and the m5.2xlarge costs $ 0.444 when hosted in London [108], their average cost in this test would not be significantly different: while the former would take $ 0.062 to execute the process in over 16 minutes, the latter would cost $ 0.065 to accomplish it in almost half the time.

7.2.2 Horizontal scalability and concurrency

Another major priority for the application was its horizontal scalability, which would allow the users to concurrently access the application and make analysis requests without performance degradation. Even though, for the cloud implementation, invoking a Lambda function and launching a new EC2 instance per request should suffice this constraint, we decided to conduct a test to ensure this characteristic. This was accomplished by sending bursts of 10, 100 and 1,000 requests and measuring their execution times. The requests consisted in finding two clusters in the `combinedDataset.csv`, using a Markov lag of one, a unitary intra-slice in-degree and one parent. Figure 7.14 graphically represents the results.

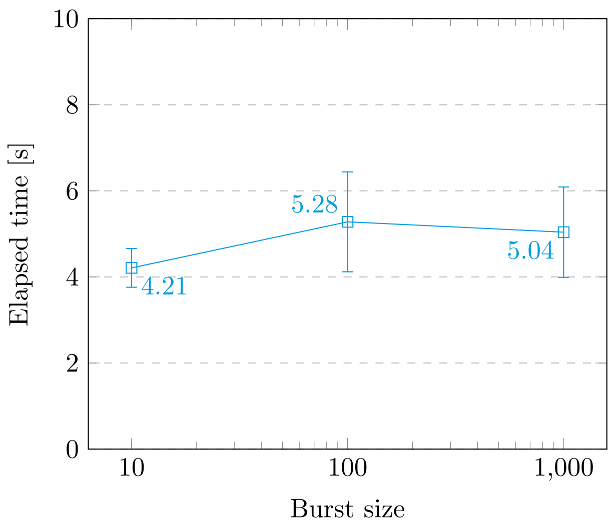


Figure 7.14: Elapsed time as a function of the burst size when clustering the `combinedDataset.csv` with a Markov lag of one, a unitary intra-slice in-degree, one parent and two clusters.

From this experiment, it is clear that the application maintains its performance despite the load and that the users will not notice whenever the servers are under pressure. As for the slightly lower duration of the smaller burst, it can be explained by the lower number of requests which is not enough to represent the amplitude of values: in fact, it falls inside the confidence interval of the remaining sizes.

7.3 Discussion

After demonstrating and appropriately stress testing the application, we should go through the requirements presented in Section 3.2 and confirm their assurance. Since the development comprised both a cloud and a local version, let us first address the prerequisites that the two address equally, and later focus on those that differ between implementations.

First off, as Section 7.1 demonstrates, QoS is ensured as the website is user friendly and extends the capabilities of the original packages with its visualisation tools, while still being transparent as the user has no perception of the multiple components involved and their interactions. What it does not depict is the notification and authentication services, both of which have been implemented, with the former sending an email upon either correct or unexpected completion of long tasks, if the user so requests, and the latter resorting to the commonly used JWTs, as explained in Sections 2.2.6 and 5.4.

Additionally, the architecture was designed to effortlessly accommodate new packages through the inclusion of their source codes and two simple interface files per functionality – detailed in Sections 5.5 and 5.6 –, as long as they can be executed in a Linux system with Python and Java installed. If that is not the case, augmenting the environment is just as simple, as it solely requires updating a Dockerfile. Besides, the users may rest assured that erroneous executions will not disturb the tool's correct execution, but will be caught by the workers and properly displayed to the practitioner, as detailed in Section 5.5 and then exemplified in Section 7.1 through a deliberate error.

Diving into implementation-specific characteristics, we can state that two of the most crucial pre-defined objectives – accepting sizeable data sets and handling both short- and long-running tasks – are accomplished in the cloud by uploading directly to S3 and launching a Lambda and an EC2 instance in parallel, as described in Sections 6.1.3 and 6.1.2 respectively. Likewise, separating files from metadata storage prevents data tables from exceeding capacity, as discussed in Section 5.6, and asynchronously executing the analyses frees processes from having to wait on lengthy requests, as argued in Section 5.4.

Furthermore, since MAESTRO is focused on clinical data interpretation, security was also one of the fundamental concerns. By distributing a local implementation where data never has to leave the premises, we give practitioners the option to improve confidentiality if they opt not to trust AWS security guarantees, which rely on AES-256 encryption [109]. Notwithstanding, it is up to each organisation to then implement HTTPS connections and secure the servers themselves from unwanted agents. This should, however, not be a concern, given that the communication with the cloud back-end uses HTTPS and that the S3 storage, the DynamoDB tables and the EC2 instances are all encrypted, with every processing instance being isolated from the others. This last assertion is quite important, as it means that in the eventuality of a hacker finding and accessing the physical server handling his request, he is still unable to retrieve anyone else's data.

As for scalability, both of its dimensions are satisfied: the two solutions support not only adding more nodes by using a message queue and a microservices architecture but also easily enhancing the existing ones by changing the instance type in the cloud architecture (as shown in Section 7.2.1) or simply upgrading the servers in the local version. Moreover, concurrency is achieved in the cloud, much like the horizontal scalability, by launching one Lambda function and one EC2 instance per processing request (without performance deterioration, as proven in Section 7.2.2). On-premises, it is up to the organisation to provide enough computing power, but since Docker Compose – an orchestration tool – was used, increasing the number of workers could not be easier, so assuring local concurrency is only a matter of having the right hardware. Besides processing, using NoSQL databases in both implementations further guarantees horizontal scalability, but now in the realm of data storage.

Finally, longevity may be evaluated by consulting web applications such as WIMP [20] or RISP [110]. When trying to access these tools' websites, the user will realise they are no longer available, demonstrating they have not withstood the test of time. While the longevity of the local platform can only be ensured by the organisations that adopt it, by using low-priced AWS services and free GitHub Pages instead of hosting our own cluster of servers, we are confident the publicly available version will remain online without much maintenance or restructuring.

Chapter 8

Conclusions

To conclude this thesis, this chapter focuses on summarising its accomplishments and outline possible future improvements that can contribute to the widespread use of the developed web application.

8.1 Achievements

In this dissertation, a language-agnostic microservice architecture for both short- and long-running web applications was devised so that practitioners no longer have to download and locally execute data analysis software. The proposed approach satisfies all of the prerequisites, including being secure, scalable and transparent while still managing to efficiently complete tasks of any duration.

This design can be adapted to incorporate any demanding data analysis command-line program and was adopted in this thesis to implement MAESTRO [7], a multivariate time series interpretation tool that makes use of dynamic Bayesian networks to impute data, detect outliers, cluster time series and predict their progression. By deploying this application using Amazon Web Services and merging the use of serverless functions with auto-scaling virtual machines, the application showed positive results regarding both concurrency and scalability.

Besides the publicly available web tool, an on-premises implementation of the introduced architecture was also developed, and its source code was made available in a GitHub repository [111]. This version can be extended with any scriptable package and is aimed at any analyst or organisation that needs this flexibility or the added layer of security that only on-premises solutions can guarantee.

With this thesis, the lack of user-friendly applications that merge machine learning and biomedicine should be closer to fulfilment, as every practitioner and organisation can now both use state-of-the-art DBN-based data analysis software online, and effortlessly host their own tools.

8.2 Future work

This final section suggests improvements and enhancements of MAESTRO that may strengthen its use by the medical and scientific community.

Bearing in mind the ease of vertically scaling the workers underlined in Section 7.2.1, it is possible to further improve the cloud implementation by choosing the EC2 instance type to launch according to the expected execution time of each incoming request. To accomplish this, the Lambda function that launches the container may consult a table that maps the input parameters and data size to an instance and the elapsed time. This table may be dynamically built by the workers when they conclude each task. Doing so will improve the application's QoS and optimise its costs.

With the advancements in homomorphic encryption, the packages could also be re-written to allow operating on top of encrypted data, which would require the front-end code to encrypt the data before uploading. However, this is only advised whenever this technology matures to the point where the added time complexity becomes negligible.

As for the on-premises implementation, the task queue may be replaced by another that allows distributing each request to the worker that is processing fewer messages. In fact, the fair dispatch used by RabbitMQ is only equitable to a certain degree: since some tasks are exponentially more demanding than others, some workers might get all of the most expensive ones. Delivering each message to the worker with the smallest number of unacknowledged messages will mitigate this issue. In case some workers are considerably less resourceful than the rest, this load balancer would not be fair as well, as these would be starved out while the others are still capable of higher loads. Solving this is a matter of limiting the number of unacknowledged messages in the more limited servers.

The monitoring and logging service of the on-premises version also has room for improvement. Even if the Docker logs are parsable, they still lack the usage and performance metrics that tools such as Amazon CloudWatch combine in a user-friendly manner. There are multiple services – the ELK stack [112] or Prometheus [113] and Grafana [114] for instance – that can be easily integrated using Docker Compose, providing a dashboard that centralises the entire system's information.

Lastly, this private dashboard could also allow to more easily upload the files regarding new packages. In this way, the tool administrators would not have to access each worker nor use command-line instructions to update the Docker Compose file and update them. Creating a fan-out queue that carries the new packages (or instructions to remove old ones) to every worker or programmatically updating the Docker images and reloading the services are just two of the methods that can help to combine this feature.

Bibliography

[1] L. V. Ho, D. Ledbetter, M. Aczon, and R. Wetzel, "The dependence of machine learning on electronic medical record quality," in *AMIA Annual Symposium Proceedings*, vol. 2017. American Medical Informatics Association, 2017, p. 883.

[2] J. T. Schwartz, M. Gao, E. A. Geng, K. S. Mody, C. M. Mikhail, and S. K. Cho, "Applications of machine learning using electronic medical records in spine surgery," *Neurospine*, vol. 16, no. 4, p. 643, 2019.

[3] M. Ghassemi, L. A. Celi, and D. J. Stone, "State of the art review: the data revolution in critical care," *Critical Care*, vol. 19, no. 1, pp. 1–9, 2015.

[4] D. S. Watson, J. Krutzinna, I. N. Bruce, C. E. Griffiths, I. B. McInnes, M. R. Barnes, and L. Floridi, "Clinical applications of machine learning algorithms: beyond the black box," *Bmj*, vol. 364, 2019.

[5] J. L. Serras, "Outlier detection for multivariate time series," Master's thesis, Instituto Superior Técnico, 11 2018.

[6] S. Arcadinho, "Model-based Learning in Multivariate Time Series," Master's thesis, Instituto Superior Técnico, 11 2018.

[7] V. Candeias. MAESTRO. Accessed: 5 December 2020. [Online]. Available: https://vascocandeias.github.io/maestro/

[8] J. Serras. METEOR. Accessed: 6 December 2020. [Online]. Available: https://jorgeserras.shinyapps.io/outlierdetection/

[9] SPSS Software — IBM. Accessed: 6 December 2020. [Online]. Available: https://www.ibm.com/analytics/spss-statistics-software

[10] Weka 3 - Data Mining with Open Source Machine Learning Software in Java. Accessed: 6 December 2020. [Online]. Available: https://www.cs.waikato.ac.nz/ml/weka/

[11] V. Candeias, S. Vinga, and A. M. Carvalho, "MAESTRO: Language-agnostic cloud architecture for short- and long-running multivariate time series analysis," 2020, in preparation.

[12] M. Ghassemi, T. Naumann, P. Schulam, A. L. Beam, I. Y. Chen, and R. Ranganath, "A Review of Challenges and Opportunities in Machine Learning for Health," *AMIA Summits on Translational Science Proceedings*, vol. 2020, p. 191, 2020.

[13] W. R. Hersh, “The electronic medical record: Promises and problems,” *Journal of the American Society for Information Science*, vol. 46, no. 10, pp. 772–776, 1995. doi: 10.1002/(SICI)1097-4571(199512)46:10¡772::AID-ASI9¿3.0.CO;2-0

[14] EMBL-European Bioinformatics Institute, “EMBL-EBI Annual Report 2019,” 11 2020.

[15] F. Doshi-Velez, Y. Ge, and I. Kohane, “Comorbidity clusters in autism spectrum disorders: An electronic health record time-series analysis,” *Pediatrics*, vol. 133, no. 1, pp. e54–e63, 1 2014. doi: 10.1542/peds.2013-0819

[16] J. A. Golden, “Deep learning algorithms for detection of lymph node metastases from breast cancer helping artificial intelligence be seen,” *JAMA - Journal of the American Medical Association*, vol. 318, no. 22, pp. 2184–2186, 12 2017. doi: 10.1001/jama.2017.14580

[17] C. J. Kelly, A. Karthikesalingam, M. Suleyman, G. Corrado, and D. King, “Key challenges for delivering clinical impact with artificial intelligence,” *BMC Medicine*, vol. 17, no. 1, p. 195, 10 2019. doi: 10.1186/s12916-019-1426-2

[18] R. H. Thompson, J. M. Kurta, M. Kaag, S. K. Tickoo, S. Kundu, D. Katz, L. Nogueira, V. E. Reuter, and P. Russo, “Tumor size is associated with malignant potential in renal cell carcinoma cases,” *The Journal of Urology*, vol. 181, no. 5, pp. 2033–2036, 2009. doi: 10.1016/j.juro.2009.01.027

[19] R. S. Sippel, D. M. Elaraj, E. Khanafshar, R. Zarnegar, E. Kebebew, Q.-Y. Duh, and O. H. Clark, “Tumor size predicts malignant potential in hürthle cell neoplasms of the thyroid,” *World Journal of Surgery*, vol. 32, no. 5, pp. 702–707, 1 2008. doi: 10.1007/s00268-007-9416-5

[20] D. Urda, J. L. Subirats, P. J. García-Laencina, L. Franco, J. L. Sancho-Gómez, and J. M. Jerez, “WIMP: Web server tool for missing data imputation,” *Computer Methods and Programs in Biomedicine*, vol. 108, no. 3, pp. 1247–1254, 12 2012. doi: 10.1016/j.cmpb.2012.08.006

[21] U. Schwiegelshohn *et al.*, “Perspectives on grid computing,” *Future Generation Computer Systems*, vol. 26, no. 8, pp. 1104–1115, 10 2010. doi: 10.1016/j.future.2010.05.010

[22] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” *National Institute of Science and Technology, Special Publication, 800-145*, 2011. doi: 10.6028/NIST.SP.800-145

[23] I. Foster, Y. Zhao, I. Raicu, and S. Lu, “Cloud Computing and Grid Computing 360-degree compared,” in *2008 Grid Computing Environments Workshop*, 2008. doi: 10.1109/GCE.2008.4738445. ISBN 9781424428601. ISSN 1424428602

[24] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge Computing: Vision and Challenges,” *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 10 2016. doi: 10.1109/JIOT.2016.2579198

[25] W. Shi and S. Dustdar, “The Promise of Edge Computing,” *Computer*, vol. 49, no. 5, pp. 78–81, 5 2016. doi: 10.1109/MC.2016.145

[26] C. Richardson, *Microservices Patterns.* Manning Publications Co., 2018. ISBN 9781617294549

[27] F. Tapia, M. Á. Mora, W. Fuertes, H. Aules, E. Flores, and T. Toulkeridis, "From Monolithic Systems to Microservices: A Comparative Study of Performance," *Applied Sciences*, vol. 10, no. 17, p. 5797, 8 2020. doi: 10.3390/app10175797

[28] M. Villamizar *et al.*, "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, (CCGrid).* Institute of Electrical and Electronics Engineers Inc., 7 2016. doi: 10.1109/CCGrid.2016.37. ISBN 9781509024520 pp. 179–182.

[29] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Colombian Computing Conference, (10CCC).* Institute of Electrical and Electronics Engineers Inc., 11 2015. doi: 10.1109/ColumbianCC.2015.7333476. ISBN 9781467394642 pp. 583–590.

[30] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, 1 2015. doi: 10.1109/MS.2015.11

[31] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984. doi: 10.1145/800217.806609

[32] Oracle. (2016) Remote Method Invocation Home. Accessed: 4 December 2019. [Online]. Available: https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

[33] A. S. Tanenbaum and R. Van Renesse, *A critique of the remote procedure call paradigm.* Vrije Universiteit, Subfaculteit Wiskunde en Informatica, 1987.

[34] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple object access protocol (SOAP) 1.1," 2000.

[35] R. T. Fielding and R. N. Taylor, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[36] J. Tihomirovs and J. Grabis, "Comparison of soap and rest based web services using software evaluation metrics," *Information Technology and Management Science*, vol. 19, no. 1, pp. 92–97, 2016.

[37] F. Halili and E. Ramadani, "Web Services: A Comparison of Soap and Rest Services," *Modern Applied Science*, vol. 12, p. 175, 2018. doi: 10.5539/mas.v12n3p175

[38] P. Giessler, M. Gebhart, D. Sarancin, R. Steinegger, and S. Abeck, "Best Practices for the Design of RESTful Web Services," in *International Conferences of Software Advances (ICSEA)*, 2015. ISBN 9781612084381 pp. 392–297.

[39] G. Mulligan and D. Gračanin, "A comparison of soap and rest implementations of a service based interaction independence middleware framework," in *Proceedings of the 2009 Winter Simulation*

Conference (WSC), 2009. doi: 10.1109/WSC.2009.5429290. ISBN 9781424457700. ISSN 08917736 pp. 1423–1432.

[40] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, p. 377–387, 6 1970. doi: 10.1145/362384.362685

[41] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '00. Association for Computing Machinery, 2000. doi: 10.1145/343477.343502. ISBN 1581131836 p. 7.

[42] R. Cattell, "Scalable SQL and NoSQL data stores," *SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011. doi: 10.1145/1978915.1978919

[43] Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases," in *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. IEEE, 2013. doi: 10.1109/PACRIM.2013.6625441 pp. 15–19.

[44] E. Rescorla, "HTTP Over TLS," Internet Requests for Comments, RFC Editor, RFC 2818, 5 2000. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2818.txt

[45] M. Jones, B. Campbell, and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants," Internet Requests for Comments, RFC Editor, RFC 7523, 5 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7523.txt

[46] D. Hardt, "The OAuth 2.0 Authorization Framework," Internet Requests for Comments, RFC Editor, RFC 6749, 10 2012. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6749.txt

[47] Federal Inf. Process. Stds. (NIST FIPS), "Announcing the advanced encryption standard (AES)," National Institute of Standards and Technology, Tech. Rep., 11 2001.

[48] P. You, Y. Peng, W. Liu, and S. Xue, "Security issues and solutions in cloud computing," in *2012 32nd International Conference on Distributed Computing Systems Workshops*, 2012. doi: 10.1109/ICDCSW.2012.20 pp. 573–577.

[49] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

[50] M. E. Zhao and Y. Geng, "Homomorphic Encryption Technology for Cloud Computing," in *Procedia Computer Science*, vol. 154. Elsevier B.V., 1 2018. doi: 10.1016/j.procs.2019.06.012. ISSN 18770509 pp. 73–83.

[51] (2006, 8) Announcing Amazon Elastic Compute Cloud (Amazon EC2) - beta. Accessed: 24 November 2020. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/

[52] S. Bhardwaj, L. Jain, and S. Jain, “Cloud computing: A study of infrastructure as a service (IAAS),” *International Journal of engineering and information Technology*, vol. 2, no. 1, pp. 60–63, 2010.

[53] K. Burkat, M. Pawlik, B. Balis, M. Malawski, K. Vahi, M. Rynge, R. F. da Silva, and E. Deelman, “Serverless containers – rising viable approach to scientific workflows,” 2020.

[54] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, “Cloud Container Technologies: a State-of-the-Art Review,” *IEEE Transactions on Cloud Computing*, 5 2017. doi: 10.1109/TCC.2017.2702586

[55] M. Boniface *et al.*, “Platform-as-a-Service architecture for real-time quality of service management in clouds,” in *5th International Conference on Internet and Web Applications and Services, ICIW 2010*, 2010. doi: 10.1109/ICIW.2010.91. ISBN 9780769540221 pp. 155–160.

[56] (2014) Introducing AWS Lambda. Accessed: 25 November 2020. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/

[57] M. Roberts and J. Chapin, *What is Serverless?* O’Reilly Media, Inc., 2017. ISBN 9781491984161

[58] Y. Ma, D. Xiang, S. Zheng, D. Tian, and X. Liu, “Moving Deep Learning into Web Browser: How Far Can We Go?” in *The World Wide Web Conference.* Association for Computing Machinery, 2019. doi: 10.1145/3308558.3313639. ISBN 9781450366748 pp. 1234–1244.

[59] World Wide Web Consortium (W3C). (2019, 12) World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation. Accessed: 7 December 2019. [Online]. Available: https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en

[60] D. Herrera, H. Chen, E. Lavoie, and L. Hendren, “WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices,” Technical Report. Technical report SABLE-TR-2018-2. Montréal, Québec, Canada: Sable Research Group, School of Computer Science, McGill University, Tech. Rep., 2018.

[61] J. L. Monteiro, S. Vinga, and A. M. Carvalho, “Polynomial-time algorithm for learning optimal tree-augmented dynamic Bayesian networks,” in *UAI*, 2015, pp. 622–631.

[62] M. Sousa and A. M. Carvalho, “Polynomial-Time Algorithm for Learning Optimal BFS-Consistent Dynamic Bayesian Networks,” *Entropy*, vol. 20, no. 4, p. 274, 2018.

[63] ——, “Learning Consistent Tree-Augmented Dynamic Bayesian Networks,” in *International Conference on Machine Learning, Optimization, and Data Science.* Springer, 2018, pp. 179–190.

[64] P. Purdom, “A transitive closure algorithm,” *BIT Numerical Mathematics*, vol. 10, no. 1, pp. 76–94, 3 1970. doi: 10.1007/bf01940892

[65] T. Leão. sdtDBN — tDBN inference and learning with static parents. Accessed: 6 December 2020. [Online]. Available: https://ttlion.github.io/sdtDBN/

[66] S. Arcadinho. learnDBN — Dynamic Bayesian Network learning. Accessed: 8 December 2020. [Online]. Available: https://ssamdav.github.io/learnDBN/

[67] R. D. Peng, "A method for visualizing multivariate time series data," *Journal of Statistical Software*, 2008. doi: 10.18637/jss.v025.c01

[68] S. Arcadinho. learnDBM — Dynamic Bayesian Multinet learning. Accessed: 8 December 2020. [Online]. Available: https://ssamdav.github.io/learnDBM/

[69] GitHub Pages — Websites for you and your projects, hosted directly from your GitHub repository. Just edit, push, and your changes are live. Accessed: 21 November 2020. [Online]. Available: https://pages.github.com/

[70] A. Stasinopoulos, C. Ntantogian, and C. Xenakis, "Commix: automating evaluation and exploitation of command injection vulnerabilities in Web applications," *International Journal of Information Security*, vol. 18, no. 1, pp. 49–72, 2019. doi: 10.1007/s10207-018-0399-z

[71] Amazon API Gateway — API Management — Amazon Web Services. Accessed: 12 November 2020. [Online]. Available: https://aws.amazon.com/api-gateway/

[72] Amazon Cognito - Simple and Secure User Sign Up & Sign In — Amazon Web Services (AWS). Accessed: 12 November 2020. [Online]. Available: https://aws.amazon.com/cognito/

[73] AWS Lambda – Serverless Compute - Amazon Web Services. Accessed: 12 November 2020. [Online]. Available: https://aws.amazon.com/lambda/

[74] Amazon EC2. Accessed: 12 November 2020. [Online]. Available: https://aws.amazon.com/ec2/

[75] Amazon SQS — Message Queuing Service — AWS. Accessed: 12 November 2020. [Online]. Available: https://aws.amazon.com/sqs/

[76] Amazon DynamoDB — NoSQL Key-Value Database — Amazon Web Services. Accessed: 12 November 2020. [Online]. Available: https://aws.amazon.com/dynamodb/

[77] Cloud Object Storage — Store & Retrieve Data Anywhere — Amazon Simple Storage Service (S3). Accessed: 25 November 2020. [Online]. Available: https://aws.amazon.com/s3/

[78] Amazon Simple Email Service — Cloud Email Service — Amazon Web Services. Accessed: 21 November 2020. [Online]. Available: https://aws.amazon.com/ses/

[79] AWS Identity & Access Management - Amazon Web Services. Accessed: 20 November 2020. [Online]. Available: https://aws.amazon.com/iam/

[80] Amazon CloudWatch - Application and Infrastructure Monitoring. Accessed: 4 December 2020. [Online]. Available: https://aws.amazon.com/cloudwatch/

[81] Y. Feng, B. Li, and B. Li, "Price competition in an oligopoly market with multiple IaaS cloud providers," *IEEE Transactions on Computers*, vol. 63, no. 1, pp. 59–73, 1 2014. doi: 10.1109/TC.2013.153

[82] R. Aljamal, A. El-Mousa, and F. Jubair, “A comparative review of high-performance computing major cloud service providers,” in *2018 9th International Conference on Information and Communication Systems (ICICS).* IEEE, 2018. doi: 10.1109/IACS.2018.8355463. ISBN 9781538643662 pp. 181–186.

[83] Amazon Web Services (AWS) - Cloud Computing Services. Accessed: 20 November 2020. [Online]. Available: https://aws.amazon.com/

[84] Cloud Computing Services — Google Cloud. Accessed: 20 November 2020. [Online]. Available: https://cloud.google.com/

[85] Cloud Computing Services — Microsoft Azure. Accessed: 20 November 2020. [Online]. Available: https://azure.microsoft.com/

[86] Cloud Application Platform — Heroku. Accessed: 20 November 2020. [Online]. Available: https://www.heroku.com/

[87] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart, “Next stop, the cloud: Understanding modern web service deployment in ec2 and azure,” in *Proceedings of the 2013 conference on Internet measurement conference*, 2013. doi: 10.1145/2504730.2504740 pp. 177–190.

[88] M. Mao and M. Humphrey, “A performance study on the VM startup time in the cloud,” in *Proceedings - 2012 IEEE 5th International Conference on Cloud Computing, CLOUD 2012*, 2012. doi: 10.1109/CLOUD.2012.103. ISBN 9780769547558 pp. 423–430.

[89] S. I. Abrita, M. Sarker, F. Abrar, and M. A. Adnan, “Benchmarking VM Startup Time in the Cloud,” in *International Symposium on Benchmarking, Measuring and Optimization.* Springer, 2018. doi: 10.1007/978-3-030-32813-9_6. ISBN 9783030328122. ISSN 16113349 pp. 53–64.

[90] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, “Peeking behind the curtains of serverless platforms,” in *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018*, 2018. ISBN 9781939133021 pp. 133–145. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-liang

[91] H. Martins, F. Araujo, and P. R. da Cunha, “Benchmarking Serverless Computing Platforms,” *Journal of Grid Computing*, 2020. doi: 10.1007/s10723-020-09523-1

[92] T. James, *The Docker Book: Containerization Is the New Virtualization.* James Turnbull, 2019. ISBN 9780988820203

[93] AWS Fargate — Serverless Compute Engine — Amazon Web Services. Accessed: 20 November 2020. [Online]. Available: https://aws.amazon.com/fargate/

[94] Cloud Run: Container to production in seconds — Google Cloud. Accessed: 20 November 2020. [Online]. Available: https://cloud.google.com/run/

[95] Build Mobile & Web Apps Fast — AWS Amplify — Amazon Web Services. Accessed: 12 November 2020. [Online]. Available: https://aws.amazon.com/amplify/

[96] R. Smith, *Docker Orchestration.* Packt Publishing Ltd, 2017. ISBN 9781787122123

[97] K. Relan, "Deploying Flask Applications," in *Building REST APIs with Flask.* Apress, 2019, pp. 159–182.

[98] T. Butler, *NGINX Cookbook.* Packt Publishing Ltd, 2017.

[99] J. Kreibich, *Using SQLite.* O'Reilly Media, Inc., 2010. ISBN 9780596521189

[100] S. Boschi and G. Santomaggio, *RabbitMQ Cookbook.* Packt Publishing Ltd, 2013. ISBN 9781849516501

[101] J. Cook, "The Dockerfile," in *Docker for Data Science.* Apress, 2017, pp. 81–101.

[102] K. Banker, *MongoDB in Action.* Manning Publications Co., 2011. ISBN 1935182870

[103] S. Bradshaw, E. Brazil, and K. Chodorow, *MongoDB: The Definitive Guide.* O'Reilly Media, Inc., 2019. ISBN 9781491954461

[104] ADNI — Alzheimer's Disease Neuroimaging Initiative. Accessed: 10 December 2020. [Online]. Available: http://adni.loni.usc.edu/

[105] S. R. Das, L. Xie, L. E. Wisse, R. Ittyerah, N. J. Tustison, B. C. Dickerson, P. A. Yushkevich, and D. A. Wolk, "Longitudinal and cross-sectional structural magnetic resonance imaging correlates of av-1451 uptake," *Neurobiology of Aging*, vol. 66, pp. 49 – 58, 2018. doi: 10.1016/j.neurobiolaging.2018.01.024

[106] M. Liu, J. Zhang, D. Nie, P. T. Yap, and D. Shen, "Anatomical landmark based deep feature representation for mr images in brain disease diagnosis," *IEEE Journal of Biomedical and Health Informatics*, vol. 22, no. 5, pp. 1476–1485, 2018. doi: 10.1109/JBHI.2018.2791863

[107] J. Mota, "Discovery of temporal patterns from multivariate time series data to support the classification of dementia profiles," Master's thesis, Instituto Superior Técnico, 2019.

[108] EC2 On-Demand Instance Pricing – Amazon Web Services. Accessed: 2 December 2020. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/

[109] K. Beer and R. Holland, "Encrypting Data at Rest," *White Paper of amazon web services*, 2014.

[110] J. Tong, P. Jiang, and Z. h. Lu, "RISP: A web-based server for prediction of RNA-binding sites in proteins," *Computer Methods and Programs in Biomedicine*, vol. 90, no. 2, pp. 148–153, 5 2008. doi: 10.1016/j.cmpb.2007.12.003

[111] V. Candeias. On-prem MAESTRO back-end. Accessed: 17 December 2020. [Online]. Available: https://github.com/vascocandeias/maestro-backend

[112] S. Chhajed, *Learning ELK stack.* Packt Publishing Ltd, 2015. ISBN 9781785887154

[113] B. Brazil, *Prometheus: Up & Running.* O'Reilly Media, Inc., 2018. ISBN 9781492034148

[114] Grafana: The open observability platform — Grafana Labs. Accessed: 4 December 2020. [Online]. Available: https://grafana.com/