On-premise containerized, light-weight software solutions for Biomedicine

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

am Fachbereich Mathematik und Informatik
der Freien Universität Berlin

vorgelegt von

Huy Le Duc

Berlin, 2023

Erstgutachter: PD Dr. Tim Conrad
Zweitgutachter: Prof. Lars Ailo Aslaksen Bongo

Tag der Disputation: 14. Juli 2023

Selbstsändigkeitserklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe. Ich erkläre weiterhin, dass ich die vorliegende Arbeit oder deren Inhalt nicht in einem früheren Promotionsverfahren eingereicht habe.

Huy Le Duc

Acknowledgements

I would like to take this opportunity to acknowledge the many individuals and institutions that have supported me in completing this dissertation. Their help and guidance have been essential to the success of this work.

First and foremost, I am grateful to my supervisor, Tim Conrad, for his unwavering support, mentorship, and constructive feedback throughout this entire process. His expertise and guidance have been crucial in shaping the direction and quality of this work. I would also like to thank the Freie Universität Berlin and the Zuse Institut for providing me with the resources and infrastructure necessary to pursue this project. Their support has been instrumental in ensuring the success of this research. I am also indebted to the many individuals who generously shared their time and expertise by participating in this study. Their contributions have been vital in achieving the research goals of this dissertation. Finally, I would like to acknowledge the support of my family and friends. Their unwavering emotional support, encouragement, and understanding have given me the strength and motivation to persevere through the challenges and obstacles of the dissertation process.

In conclusion, I am grateful to everyone who has supported me throughout this dissertation. I hope that this work will contribute to the knowledge and understanding of Containerized Software Solutions for Biomedicine, and I am honored to have had the opportunity to undertake this endeavor.

Abstract

Bioinformatics software systems are critical tools for analysing large-scale biological data, but their design and implementation can be challenging due to the need for reliability, scalability, and performance. This thesis investigates the impact of several software approaches on the design and implementation of bioinformatics software systems. These approaches include software patterns, microservices, distributed computing, containerisation and container orchestration. The research focuses on understanding how these techniques affect bioinformatics software systems' reliability, scalability, performance, and efficiency. Furthermore, this research highlights the challenges and considerations involved in their implementation. This study also examines potential solutions for implementing container orchestration in bioinformatics research teams with limited resources and the challenges of using container orchestration. Additionally, the thesis considers microservices and distributed computing and how these can be optimised in the design and implementation process to enhance the productivity and performance of bioinformatics software systems. The research was conducted using a combination of software development, experimentation, and evaluation. The results show that implementing software patterns can significantly improve the code accessibility and structure of bioinformatics software systems. Specifically, microservices and containerisation also enhanced system reliability, scalability, and performance. Additionally, the study indicates that adopting advanced software engineering practices, such as model-driven design and container orchestration, can facilitate efficient and productive deployment and management of bioinformatics software systems, even for researchers with limited resources. Overall, we develop a software system integrating all our findings. Our proposed system demonstrated the ability to address challenges in bioinformatics. The thesis makes several key contributions in addressing the research questions surrounding the design, implementation, and optimisation of bioinformatics software systems using software patterns, microservices, containerisation, and advanced software engineering principles and practices. Our findings suggest that incorporating these technologies can significantly improve bioinformatics software systems' reliability, scalability, performance, efficiency, and productivity.

Zusammenfassung

Bioinformatische Software-Systeme stellen bedeutende Werkzeuge für die Analyse umfangreicher biologischer Daten dar. Ihre Entwicklung und Implementierung kann jedoch aufgrund der erforderlichen Zuverlässigkeit, Skalierbarkeit und Leistungsfähigkeit eine Herausforderung darstellen. Das Ziel dieser Arbeit ist es, die Auswirkungen von Software-Mustern, Microservices, verteilten Systemen, Containerisierung und Container-Orchestrierung auf die Architektur und Implementierung von bioinformatischen Software-Systemen zu untersuchen. Die Forschung konzentriert sich darauf, zu verstehen, wie sich diese Techniken auf die Zuverlässigkeit, Skalierbarkeit, Leistungsfähigkeit und Effizienz von bioinformatischen Software-Systemen auswirken und welche Herausforderungen mit ihrer Konzeptualisierungen und Implementierung verbunden sind. Diese Arbeit untersucht auch potenzielle Lösungen zur Implementierung von Container-Orchestrierung in bioinformatischen Forschungsteams mit begrenzten Ressourcen und die Einschränkungen bei deren Verwendung in diesem Kontext. Des Weiteren werden die Schlüsselfaktoren, die den Erfolg von bioinformatischen Software-Systemen mit Containerisierung, Microservices und verteiltem Computing beeinflussen, untersucht und wie diese im Design- und Implementierungsprozess optimiert werden können, um die Produktivität und Leistung bioinformatischer Software-Systeme zu steigern. Die vorliegende Arbeit wurde mittels einer Kombination aus Software-Entwicklung, Experimenten und Evaluation durchgeführt. Die erzielten Ergebnisse zeigen, dass die Implementierung von Software-Mustern, die Zuverlässigkeit und Skalierbarkeit von bioinformatischen Software-Systemen erheblich verbessern kann. Der Einsatz von Microservices und Containerisierung trug ebenfalls zur Steigerung der Zuverlässigkeit, Skalierbarkeit und Leistungsfähigkeit des Systems bei. Darüber hinaus legt die Arbeit dar, dass die Anwendung von Software-Engineering-Praktiken, wie modellgesteuertem Design und Container-Orchestrierung, die effiziente und produktive Bereitstellung und Verwaltung von bioinformatischen Software-Systemen erleichtern kann. Zudem löst die Implementierung dieses Software-Systems, Herausforderungen für Forschungsgruppen mit begrenzten Ressourcen. Insgesamt hat das System gezeigt, dass es in der Lage ist, Herausforderungen im Bereich der Bioinformatik zu bewältigen und stellt somit ein wertvolles Werkzeug für Forscher in diesem Bereich dar. Die vorliegende Arbeit leistet mehrere wichtige Beiträge zur Beantwortung von Forschungsfragen im Zusammenhang mit dem Entwurf, der Implementierung und der Optimierung von Software-Systemen für die Bioinformatik unter Verwendung von Prinzipien und Praktiken der Softwaretechnik. Unsere Ergebnisse deuten darauf hin, dass die Einbindung dieser Technologien die Zuverlässigkeit, Skalierbarkeit, Leistungsfähigkeit, Effizienz und Produktivität bioinformatischer Software-Systeme erheblich verbessern kann.

Deliverables

1. A framework for distributed computing with state-of-the-art technologies:
   (a) A revised definition of distributed computing and related concepts in the context of architecture, based on a comprehensive literature review.
   (b) A review of deployment measurement approaches and an illustration of distributed computing in terms of scope, configuration, and mode.
   (c) A classification and taxonomy of distributed computing solutions with containers, including a discussion of existing container technologies and their relation to microservices.
   (d) Identification of open issues and research challenges in the field of distributed computing for bioinformatics.
2. An approach to building and managing a distributed computing framework:
   (a) A definition of software architecture and its components, incorporating state-of-the-art software patterns for utilities, based on an analysis of current best practices in the field.
   (b) A prototype implementation of the proposed approach, including a detailed description of the design and implementation process, as well as an evaluation of its functionality and usability.
   (c) A framework using state-of-the-art architectural software patterns, which can serve as a blueprint for other researchers and practitioners in the field of bioinformatics.
3. A demonstration of the framework's capabilities:
   (a) A comprehensive evaluation of the approach in terms of performance, cost, and resource optimization, using a variety of datasets and benchmarking methods.
   (b) A scalable approach to process execution, which can handle large-scale datasets and complex computational tasks.
   (c) A flexible framework for integrating new tools and methods, which can adapt to the evolving needs of bioinformatics research teams.

Contents

Chapter 1

Introduction

In the biomedical field, the vast amount of data generated through various experiments and studies presents a major challenge for researchers. Effective management and analysis of this data are crucial for extracting valuable insights and informing decision-making. Data engineering, which encompasses a range of fields including machine learning, statistics, and bioinformatics, provides the necessary tools and techniques for collecting, storing, processing, and analysing this data [28, 43, 47, 107]. As such, it plays a central role in the biomedical field.

However, data engineering also significantly overlaps with software engineering, as it involves the development of systems and tools to manage and analyse data. In the biomedical field, these systems and tools are often complex. Furthermore, it may require integration with various data sources and analysis pipelines. As a result, a strong foundation in software engineering principles is crucial for the successful design and implementation of data engineering solutions in the biomedical field. Specifically, data engineering is a vital field within the realm of software engineering that encompasses the modelling, design, access, control, and evaluation of data, as well as the deployment, evolution, maintenance, and standardisation of data engineering systems using existing and emerging technologies [140]. It is a multifaceted domain that plays a crucial role in acquiring knowledge; moreover, it enables new insights into existing and new data. Software system architecture is a fundamental aspect of data engineering, as it can significantly influence the execution of tasks. A well-designed software system architecture can improve a data engineering system's efficiency, scalability, and maintainability in the biomedical field.

One of these promising software architectures is distributed computing. Distributed computing has proven to be a promising software architecture for handling large computing tasks. It is commonly integrated into data analysis platforms, which use a predefined software architecture to manage and process data. The use of distributed computing offers a range of benefits, including scalability, reliability, security, high maintainability, testability, and deployability. In biomedical analysis, distributed computing is a particularly useful tool. However, privacy concerns and the diversity of data types must be carefully considered when selecting the appropriate tool. As such, the design of the software architecture must reflect these

requirements to be effective. In summary, software architecture is central to using distributed computing for biomedical analysis.

Despite the numerous advantages of distributed computing and the significant interest it has garnered from both the research and industrial sectors, it remains a highly challenging field of study. This circumstance is due to the wide range of implementations, architectural approaches, and deployment options utilised in distributed systems, each of which may present unique challenges and trade-offs. In particular, the complexity, accessibility, and overall data flow of a given system must be carefully considered to leverage the benefits of distributed computing effectively. These factors contribute to the continued complexity of this field and the need for ongoing research and development.

This thesis comprehensively investigates the software architectural considerations in constructing a distributed bioinformatics system. This research focuses on the design patterns for biomedical data analysis and the development of a framework that combines distributed computing with container-based virtualisation. The proposed framework is intended to facilitate the efficient and effective management of large-scale data analysis tasks within the healthcare field. The contributions of this thesis have been significant in developing a distributed analysis system for omics data and have the potential to impact the field of bioinformatics and healthcare. The proposed framework offers a scalable and flexible approach to data analysis that is adaptable to the ever-evolving needs of the bioinformatics community.

This research was conducted with the support of the German Ministry for Education and Research (BMBF) through the Berlin Big Data Center and the Berlin Center for Machine Learning (01IS14013A and 01IS18037J) as well as the Forschungscampus MODAL (project grant 3FO18501). The developed software was tested in the context of the MEDLAB project, which aims to advance the capabilities of data analysis in healthcare through the use of next-generation technologies.

1.1 Scope and Challenges

In recent years, the volume of biomedical data has been rapidly increasing [56]. This data can come in a wide range of types, including but not limited to CT data

and text-based databases. The sensitivity of this data, which can include personal medical information, also requires that it be stored in a manner that adheres to relevant regulations [141]. As such, the storage of biomedical data is a critical concern for those in the field. Furthermore, the analysis of these data often involves the use of multiple tools, each of which may have its dependencies. These dependencies can lead to conflicts if all the tools are installed on a single machine, which can be a time-consuming and tedious task for system administrators to manage [55]. In order to minimise the effort required for system configuration and dependency management, it is important for the software system setup to be as simple and streamlined as possible.

This section will delve into the various challenges faced in the software, data, and biomedical fields, particularly regarding the complexity and diversity of biomedical data. These challenges necessitate developing innovative solutions for the adequate storage, management, and analysis of this data. In order to provide a comprehensive overview of these challenges, we will expand upon the current state of the art in the biomedical data storage and analysis field in Chapter *2 State of the art*. However, before delving into the state of the art, it is essential to first outline these challenges in this section. Moreover, we outline the challenges when ensuring that biomedical data is handled in a manner that is both efficient and effective while also considering the privacy implications of handling such sensitive data.

Software Challenges

Building software for biomedicine involves a range of challenges, including understanding and managing complex system requirements, defining the scope of the application, and addressing security and performance concerns. These challenges are amplified since bioinformatics research often faces limited resources, including time and expertise, in software development, which can include setting up and maintaining the software system. Furthermore, in the fast-paced field of biomedicine, integrating new tools and technologies are often necessary, which can be challenging due to the highly regulated nature of the deployment environment and the use of diverse programming languages and frameworks. Each of these challenges significantly affects the others and improvements in one area can impact other aspects of

the project. In addition, large-scale computing infrastructure is increasingly essential for scientific projects and must also address the general challenges of software development. In summary, developing software for biomedicine requires addressing technical, regulatory, also interdependent challenges to succeed.

Data Challenges

The biomedical data analysis framework must address several data volume, variety, and velocity challenges. The volume of data refers to the amount of data being collected, which serves as the foundation for further analysis. Velocity, on the other hand, refers to the speed at which data is generated and moves, which is a crucial consideration in biomedical data analysis, where fast data streams are often needed for making informed decisions. Variety, in turn, refers to the diverse types of data that may be collected, including patient data and protein sequences. This diversity poses a challenge in terms of standardising and distributing all of the collected data.

There has been a shift in data engineering concerns in the field of biomedical data analysis, particularly in the realm of high-throughput sequencing. In the past, the main challenge was generating new data, but the focus has now shifted to handling large amounts of data, with the primary challenge being the analysis of said data. This shift is depicted in Figure 1.1. Overall, the shift from data generation to data handling has resulted in a need for software systems that can effectively manage large volumes of data.

Medical Challenges

The analysis of medical data presents a unique set of challenges due to the diverse and often unstructured nature of the data. Medical data can include unstructured data, which is disorganised and stored in various formats, semi-structured data, which has some organisation but is not stored in a specialised repository, and structured data, which is formatted and stored in a specialised repository. Finding a cohesive approach for analysing all types of medical data is a the challenge, and often requires the creation of additional structures to organise the data.

Another challenge in biomedical data analysis is the recognition of patterns in

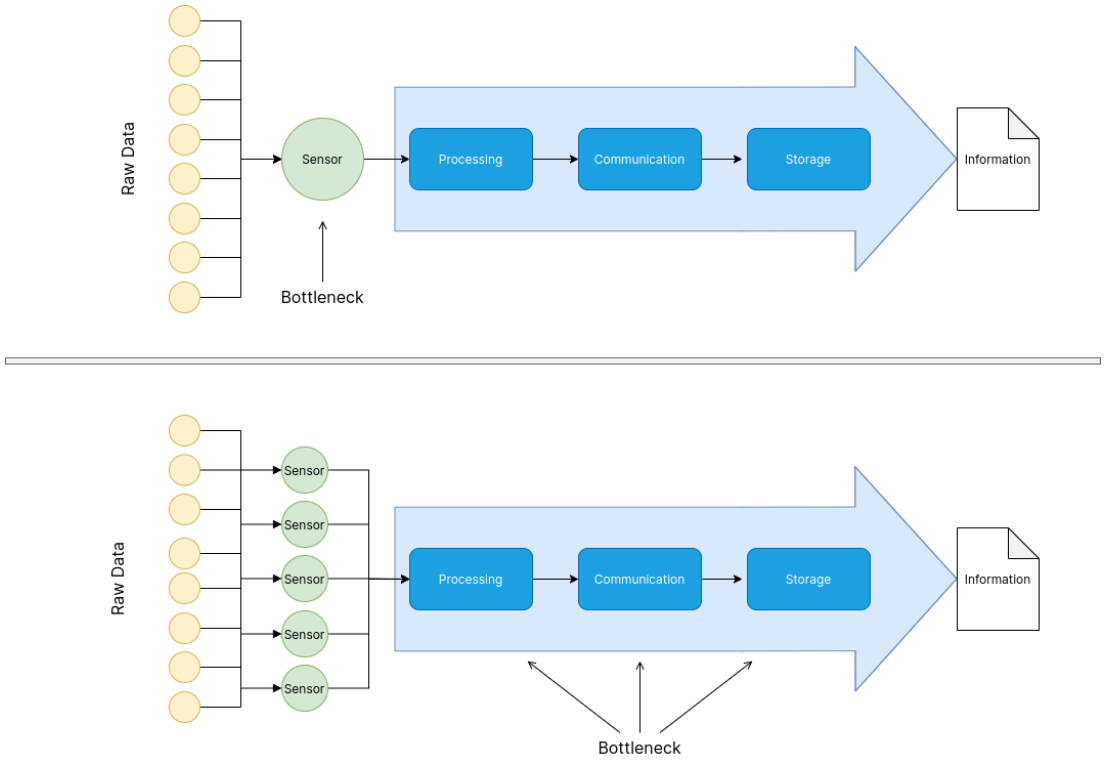


Figure 1.1: A comparison of two data processing pipelines is shown. In the top pipeline, a single sensor is responsible for processing multiple inputs, leading to a bottleneck at the sensor stage. Conversely, the bottom pipeline utilises multiple sensors to collect raw data, shifting from data generation to data handling. While this shift in the bottleneck from the sensor to the processing part may increase the overall system's efficiency. It also presents a new challenge as solutions must now be found for effectively handling the increased volume of data at the processing stage.

the data. Human experts can identify valuable insights through pattern recognition, but this process can take time to replicate programmatically. While computer-aided analysis offers the potential to analyse biomedical data in ways that surpass human capabilities, there are still challenges in using these methods for pattern recognition.

In addition to these technical challenges, the handling of personal medical data raises significant concerns about data protection and access control. Federal guidelines for personal data processing in the medical field provide strict regulations for data security and access, and the use of third-party infrastructure is often restricted or prohibited. The sensitivity of medical data and the need for careful regulation in its handling highlight the importance of addressing these challenges in biomedical data analysis.

In conclusion, the fields of software, data, and biomedical face a range of challenges that must be addressed in order to effectively store and analyse biomedical data. These challenges include understanding and managing complex system requirements in software development, addressing the volume, variety, and velocity of data in data analysis, and dealing with regulatory concerns in the biomedical field. These challenges must be addressed in order to ensure that biomedical data is handled in a manner that is both efficient and effective.

1.2 Research problems and objectives

The primary objective of this thesis is to investigate comprehensively the potential of distributed computing technologies for the analysis of biomedical data. Through an in-depth exploration and demonstration of these technologies, this study aims to contribute to the advancement of the field and provide a practical approach to the adequate support of large and complex datasets in the realm of biomedical analysis. We summarise this objective in this research question:

> **How can a multifaceted biomedical data analysis system with a distributed architecture be set up and managed across single or multiple providers, considering software engineering guidelines, virtualisation, data management, and related aspects for small and medium-sized research teams?**

This research investigates the use of distributed computing technologies in biomedical data analysis. Specifically, we will focus on issues such as software patterns, the design and implementation of bioinformatics systems using Docker and microservices, the integration of data from multiple sources, and the scalability and performance of these systems. We have summarised our research questions as follows.

- How does the use of software patterns in the design of a bioinformatics software system impact its reliability, scalability, and performance?
- What are the key considerations and challenges in designing and implementing a bioinformatics software system with Docker, microservices, and distributed computing?
- How does the integration of advanced software engineering principles and practices into a bioinformatics software system support the effective integration of data from multiple sources?
- How does the adoption of a distributed architecture for a bioinformatics software systems impact the efficiency and productivity of a small research group?
- What are the challenges and potential solutions for the implementation of container orchestration in bioinformatics research teams with limited resources, and what are the benefits and trade-offs of using container orchestration in such a context?
- What are the key factors that influence the success of a bioinformatics software system with Docker, microservices, and distributed computing, and how these can be optimised in the design and implementation process?
- How does the implementation of parallel computation techniques in our framework improve the speed of data analysis in bioinformatics?
- What are the most effective approaches for integrating and analysing data from multiple sources within our framework in bioinformatics?
- How can our framework be scaled to handle effectively large-scale datasets in bioinformatics, and how can improve machine learning techniques are applied to optimise data integration and analysis in this context?

Objective

This thesis aims to investigate the use of distributed computing technologies for biomedical data analysis. The research objectives include examining the definition and essential characteristics of distributed computing, exploring the approaches and mechanisms used in distributed computing, and understanding the impact of software patterns on the reliability, scalability, and performance of a bioinformatics software system. The research will also focus on the key considerations and challenges in designing and implementing a bioinformatics software system using Docker, microservices, and distributed computing, and will investigate the impact of a distributed architecture on the efficiency and productivity of a small research group. Additionally, the study will identify the key factors that influence the success of a bioinformatics software system using these technologies and will examine the effectiveness of parallel computation techniques and the most effective approaches for integrating and analysing data from multiple sources within a bioinformatics software system. Finally, the research will investigate how a bioinformatics software system can be scaled to handle large-scale datasets and how machine learning techniques can be applied to optimise data integration and analysis.

1.3 Evaluation method

In this thesis, we developed an approach to handle distributed computing for biomedical data analysis. We evaluated this approach with various experimental platforms with different hardware specifications. We executed the experiments on an infrastructure set: Freie University, Zuse Cluster, and personal machines. We adjusted the parameters for each experiment execution. We list the platforms and hardware specifications used for our approaches.

- Cloud orchestrators: Docker swarm
- Docker technologies: Docker engine, Docker-compose, Docker swarm
- Applications: Nginx, Minio, Postgres
- Monitoring: Docker stats, cAdvisor

- Workload generators: javascript
- OS: Linux-based systems

1.4 Contribution and findings

In this thesis, we propose a distributed computing framework, distributed computing, for biomedical data analysis that addresses challenges in software architecture, requirements management, and virtualisation. Our contribution aims to solve the research questions defined in section 1.1 and support the workflow for biomedical data analysis with state-of-the-art technologies such as virtualisation. The main contributions of this thesis are:

1. A framework for distributed computing with state-of-the-art technologies:
    - A revised definition of distributed computing and related concepts in the context of architecture
    - A review of deployment measurement approaches
    - An illustration of distributed computing in terms of scope, configuration, and mode
    - A classification and taxonomy of distributed computing solutions with containers
    - A discussion of existing container technologies and their relation to microservices
    - Identification of open issues and research challenges
2. An approach to building and managing a distributed computing framework:
    - A definition of software architecture and its components, incorporating state-of-the-art software patterns for utilities
    - A prototype implementation of the proposed approach
    - A framework using state-of-the-art architectural software patterns
3. A demonstration of the framework's capabilities:

- A comprehensive evaluation of the approach in terms of performance, cost, and resource optimisation
- A scalable approach to process execution
- A flexible framework for integrating new tools and methods

In addition, we present a software design for scientific biomedical systems using Linux containerised virtualisation to create independent software environments and prevent dependency conflicts. Tasks can be executed on a distributed system to parallelise computational effort with this containerised architecture. Our system also exposes its service application programming interface using the representational state transfer architecture based on hypertext transfer protocol, allowing access as a web service. This design is adaptable to new demands and not limited to specific biomedical applications.

Our findings highlight the potential for the use of advanced software engineering practices, such as software patterns, microservices, and Docker containers, in the development of bioinformatics software systems. These technologies can significantly improve the system reliability, scalability, and performance, and facilitate the efficient integration and management of multiple types of data. The adoption of a distributed architecture was also found to enhance the efficiency and productivity of a small research group through the parallelisation of tasks and the utilisation of additional resources. Our system demonstrated its effectiveness in handling large datasets and reducing computational time, making it a valuable tool for bioinformatics research. Overall, these findings suggest that the integration of modern software engineering practices into bioinformatics software systems can significantly improve the efficiency and productivity of research groups in this field.

1.5 Outline

The main objective of this study is to investigate and improve the the current state of bioinformatics software engineering. To achieve this the goal in chapter *2 State of the art*, we first present a review of the existing literature in this field, including the foundations of bioinformatics software design and the most relevant related work. We

specifically highlight the importance of distributed computing and containerisation and identify the main challenges and open issues in both software and biomedical domains.

The chapter *3 Advanced Software Engineering in Bioinformatic: A Case Study of Design and Implementation* of the thesis presents a case study of advanced software engineering in bioinformatics, with a focus on the design and implementation of a new system. We discuss the technical considerations that guided the design of the system in detail, including topics such as technical debt, software architecture, and the use of cloud technologies and containerisation. We then describe the proposed architecture, which is based on microservices and representational state transfer, and discuss the implementation of the system, including the decomposition of services by domain and the use of container orchestration.

The chapter *4 Enhancing Bioinformatics Analysis with Our Proposed System: Real-World Applications and Implications* evaluates the real-world applications and implications of the proposed system. After examining the system's performance features, the chapter presents biomedical experiments demonstrating its capabilities. The chapter concludes with a discussion of the results, including interpretation and context, limitations and bias, and comparison to other works. It also provides a summary of the main conclusions and their implications.

The final chapter *5 Discussion: A comparative analysis of summary, limitations, and comparison* discusses the results and conclusions of the study. After a summary of the main findings, the chapter explores the limitations and bias of the study and compares the results to those of other relevant works. Finally, the chapter concludes with a discussion of the main conclusions, their implications for future research, and suggestions for further investigation.

In summary, this thesis makes a significant contribution to the field of bioinformatics software engineering by presenting a comprehensive and practical approach to the design and implementation of systems that can effectively support the analysis of large and complex datasets.

Chapter 2

State of the art

This chapter reviews distributed computing solutions and provide an overview of containerisation, a technical implementation in virtualisation. We review and outline the proposed approaches in the form of a literature review. Afterwards, this chapter discusses issues and research challenges in biomedical data analysis concerning software development, data management and general biomedical challenges.

2.1 Introduction

In recent years, distributed computing and containerisation have gained significant attention as critical technologies for addressing the challenges of managing and analysing large amounts of data in the field of bioinformatics. Distributed computing refers to the use of multiple computers to perform a task, often involving the coordination of data and computation across a network of machines. Containerisation, on the other hand, is a technique for virtualisation that involves the packaging of an application and its dependencies into a single container, enabling the application to be easily deployed and ran on any platform.

In this chapter, we review the literature on software and data management systems for bioinformatics data analysis that uses these technologies. We begin by providing a background on distributed computing and containerisation, highlighting their key features and benefits. We then survey the approaches and techniques proposed in the literature for designing and implementing software systems for bioinformatics data analysis using Docker, microservices, distributed computing and data management. This survey includes software patterns, advanced engineering principles and practices, and distributed architectures.

We identify a number of open research questions and challenges related to the design and implementation of bioinformatics software systems with Docker, microservices, and distributed computing. These include questions about the impact of software patterns on reliability, scalability, and performance of a bioinformatics software system; the key considerations and challenges in designing and implementing such a system; the role of advanced software engineering principles and practices in supporting the effective integration of data from multiple sources; and the factors that influence the success of such a system and how they can be optimised in the

design and implementation process. We also explore the potential benefits and challenges of applying parallel computation techniques and machine learning techniques to improve the speed and effectiveness of data analysis, and the scalability of our framework for large-scale handling datasets in bioinformatics.

This chapter aims to provide a comprehensive review of the state of the art in the design and implementation of software systems and data management systems for bioinformatics data analysis using Docker, microservices, and distributed computing. We hope that this review will help to identify the critical research questions and challenges that need to be addressed in order to advance the field and support the effective integration and analysis of data from multiple sources in bioinformatics.

2.2 Background: Foundations of bioinformatics software design

2.2.1 Bioinformatics software systems

Bioinformatics software systems are an essential component of modern research in the field of biology. These systems allow researchers to analyse and interpret extensive and complex datasets, enabling them to extract knowledge and insights that would not be possible using traditional methods. Many different types of bioinformatics software systems are available, each designed to meet specific needs and goals. Some examples of bioinformatics software systems include:

Data integration and management systems, such as GeneMania, a web-based platform that enables researchers to collect, integrate, and analyse data on gene and protein interactions. These systems enable researchers to efficiently and effectively manage large volumes of data from multiple sources.

Workflow management systems, such as Galaxy, which is an open-source a platform that enables researchers to design and execute complex analysis pipelines using a wide range of tools and resources. These systems allow researchers to coordinate the execution of multiple tools and processes in a systematic and reproducible way, ensuring the reliability and robustness of their analyses.

Data visualisation tools, such as Cytoscape, which is an open-source a platform that enables researchers to visualise and analyse complex biological networks and pathways. These tools allow researchers to explore and understand large datasets by generating interactive graphs, plots, and other visualisations of the data.

Machine learning and data mining tools, such as WEKA, which is an an open-source platform that provides a suite of machine learning algorithms for data mining and predictive modelling. These tools enable researchers to discover patterns and insights in large datasets that might need to be apparent using traditional approaches.

Distributed computing frameworks, such as Apache Hadoop, which is an an open-source platform that enables researchers to process and analyse large datasets using a distributed computing approach. These frameworks allow researchers to scale their analyses to large datasets and perform them more efficiently, enabling them to tackle problems that would be intractable using traditional methods.

These bioinformatics fields have various focuses that depend on each research use case. In a later section, we will expand on the open issues that challenge their collection of fields.

2.2.2 Distributed computing

Definiton and related terms

For distributed exist various definitions [70, 172]. Here we define a distributed system as a set of autonomous computers grouped into a single system from a user perspective. Instead of computers, we also use the term nodes or processors. Distributed systems use a collection of independent protocols To achieve this grouping. Generally, this collection is a software layer between the operating system and the distributed application. This software layer is also referred to as a middleware layer. The middle layer provides rules and procedures to ensure communication, reliability and transaction.

The term Distributed systems shares connect to related other terms. These terms include concurrent systems and parallel systems. A system can be characterised as parallel and distributed at the same time. We outline the nuanced viewpoints based on two points of view on how to differentiate parallel and distributed computing.

Kshemkalyani et al. [96] and Ghosh [70] described one defining aspect concerning parallel systems as shared memory. Kshemkalyani et al. add that in this architecture, the access latency for memory is the same as for any processor. We illustrated the shared memory concept in figure 2.1. This architecture is also referred to as uniform memory access architecture. From another perspective, Peleg argues that another defining characteristic of parallel and distributed systems is the coupling level. The coupling level has two main forms: tight and loose. "In a tightly coupled system (e.g., a parallel machine), the processors typically work in tight synchrony, share memory to a large extent and have high-speed and reliable communication mechanisms between them. In contrast, in a loosely coupled distributed system (e.g., a wide-area communication network) the processors are more independent than tight coupled, communication is less frequent and less synchronous, and cooperation is more limited. " [137]. Depending on the granularity of coupling, computing systems could be characterised as parallel or distributed. Therefore depending on the point of view, terms and functions bleed into each other.

The situation is further complicated by the traditional uses of the terms parallel and distributed algorithm that do not quite match the above definitions of parallel and distributed systems (see below for a more detailed discussion). Nevertheless, as a rule of thumb, high-performance parallel computation in a shared-memory multiprocessor uses parallel algorithms, while the coordination of a large-scale distributed system uses distributed algorithms.[21]

Design goals for distributed systems include sharing resources and ensuring openness. In addition, designers aim to hide many of the intricacies related to the distribution of processes, data, and control. However, this distribution transparency not only comes at a performance price but in practical situations, it can only partially be achieved. The fact that trade-offs need to be made between achieving various forms of distribution transparency is inherent to the design of distributed systems and can easily complicate their understanding. One specific challenging design goal that only sometimes fits well with achieving distribution transparency is scalability. This matter is particularly true for geographical scalability, in which case hiding latencies and bandwidth restrictions can turn out to be complicated. Likewise, administrative scalability, by which a system is designed to span multiple administrative domains,

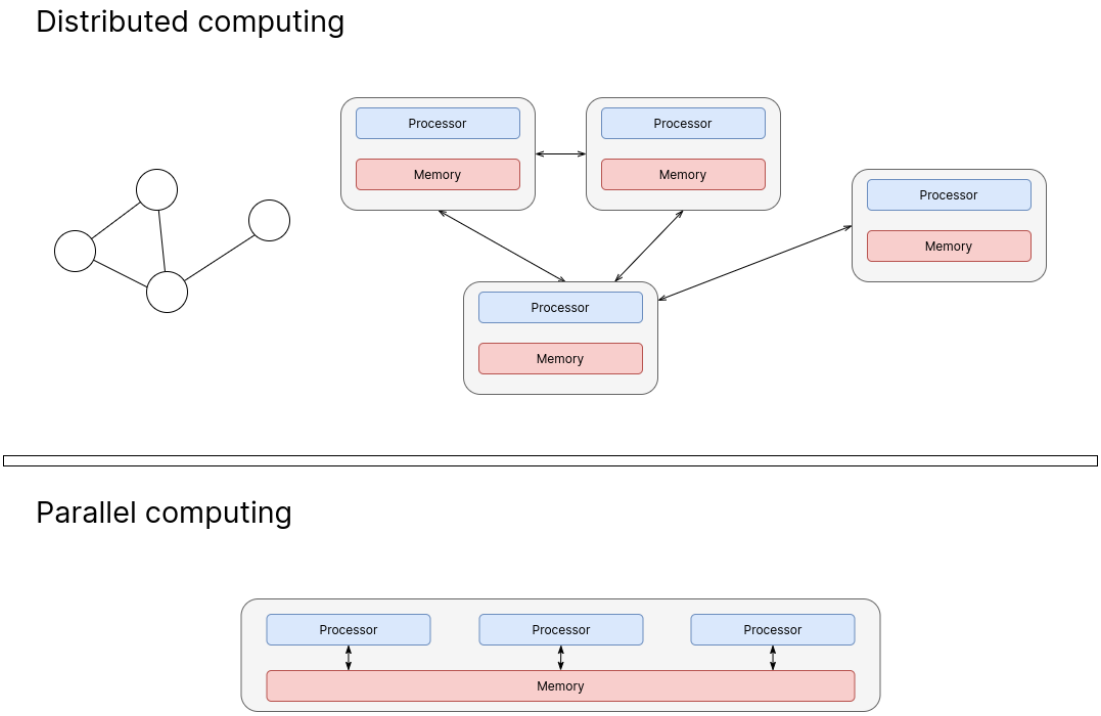


Figure 2.1: The figure on the right illustrates the difference between distributed and parallel systems. Figure (a) is a schematic view of a typical distributed system; the system is represented as a network topology in which each node is a computer, and each line connecting the nodes is a communication link. Figure (b) shows the same distributed system in more detail. Each computer has its local memory; information can be exchanged only by passing messages from one node to another using the available communication links. Figure (c) shows a parallel system in which each processor has direct access to a shared memory. The difference between distributed and parallel computing

may easily conflict with goals for achieving distribution transparency. Matters are further complicated because many developers initially make assumptions about the underlying network that need to be corrected. Later, when assumptions are dropped, it may be difficult to mask unwanted behaviour. A typical example is assuming that network latency is insignificant. Other pitfalls include assuming the network is reliable, static, secure, and homogeneous.

Different types of distributed systems exist, which can be classified as being oriented toward supporting computations, information processing, and pervasiveness. Distributed computing systems are typically deployed for high-performance applications often originating from the field of parallel computing. A field that emerged from parallel processing was initially grid computing with a strong focus on the worldwide sharing of resources, in turn leading to what is now known as cloud computing. Cloud computing goes beyond high-performance computing and also supports distributed systems found in traditional office environments, where we see databases playing an important role. Typically, transaction processing systems are deployed in these environments. Finally, an emerging class of distributed systems is where components are small, the system is composed in an ad hoc fashion, but most of all, is no longer managed through a system administrator. Pervasive computing environments, including mobile-computing systems and sensor-rich environments, typically represent this last class.

Taxonomy

Distributed computing uses different strategies, methods and techniques. So that, different classifications implement their solutions in alignment with their desired characteristic. We investigated academic and industrial solutions and proposed a classisfication based on figure 2.2 . it is an extended classification from [114,137,162, 172].

The following sections outline the details of each technology and characteristic. We classify the solutions according to type, design goals, architecture and middleware organisation.

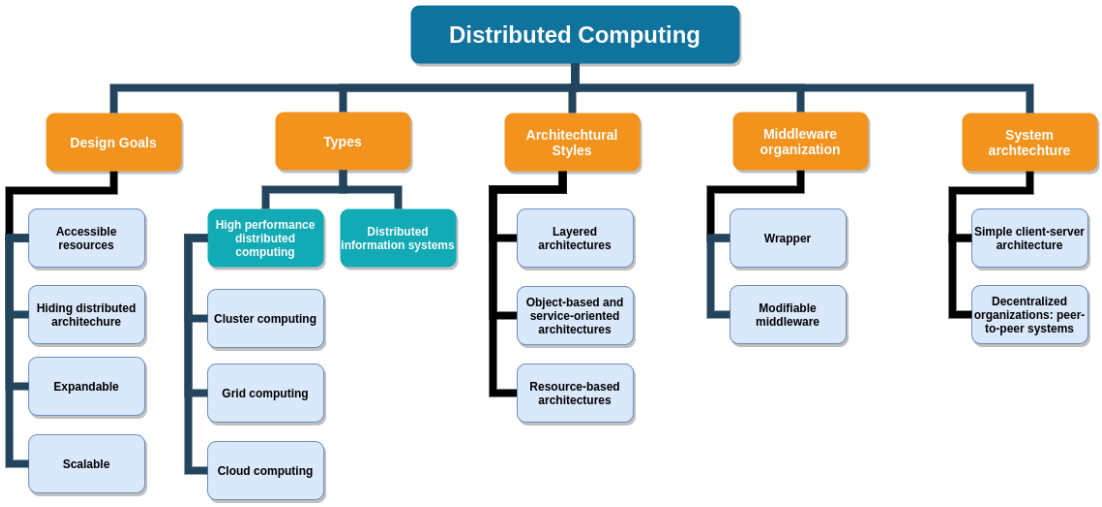


Figure 2.2: Illustration of distributed taxonomy

Design goals

In this section, we discuss four key goals in distributed computing. These goals underline the benefits of distributed computing architecture. Generally, the four goals include

- easily accessible resources
- hiding distributed architecture to the end user
- it should be open
- be able to scale according to demand

Supporting resource sharing One of a user's main distributed computing goals is to facilitate access and share remote resources. These resources can range from data, files, services, storage and similar resources. The reasons for sharing resources vary depending on the requirements. For example, from a biomedical perspective, it is easier to maintain and manage a sizeable reliable system than having to manage each resource individually and manually. Furthermore, collaboration and information exchange is improved since the application/user can access the resources directly. This connectivity allows geographically or department-separated users to collaborate with shared resources [172, 180].

Hiding distributed architecture to the user Another important distributed computing goal is to present the system as a single uniform entity, i.e. hiding the distributed architecture from the user. The International organisation for Standardisation refers to this obfuscation as transparency [85].

There are diffrent types of transparancy. Three main types are: Location, concurrency and access. First, Location transparency obscures the process or resource location of the user. This location transparency uses naming to assign only addresses or names to resources. The uniform resource locator (URL) only points to a resource. So the user needs to be aware of when resources or processes move from the data centres. Second, concurrency transparency hides simultaneous access to processes and resources from the user. One drawback in sharing resources is users requesting and altering data simultaneously. Distributed computing intends to establish a consistent state for the provided resources. There are several possibilities to achieve consistency, e.g. locking access with exclusive permission or transactions. Third, access transparency obfuscates the data implementation. In other words, access transparency hides the data architecture from the user. Like the naming convention for location transparency, the user only views the designated name. The distributed system handles communication and process on the underlying systems.

Being open The next goal for distributed computing is to allow other systems to use or integrate its resource or services into other systems. Generally, a distributed system includes and integrates components from various origins. In other words, distributed systems aim to be open.

Distributed computing openness relies on rules for operation and communication on agreed-upon syntax and semantics. These rules are part of an interface. This interface defines the general condition for using the service, e.g. function name, function parameter, return values, possible exception and other conditions. With a well-defined interface various component implementations can be exchanged easily. This concept is also referred to as interoperability. The interface handles a standardised mode of communication.

An adjacent design goal to interoperability is portability. Portability refers to deploying the whole distributed system or components on different systems. This

deployment requires established interfaces. Blair and Stefani added that the independence of component implementation is essential for interoperability and Portability [29].

The final design goal concerning openness is extensibility. With extensibility Distributed system can integrate or replace components without affecting existing ones, e.g. run on a different operating system or change/add databases. This flexibility relies on the definition of interfaces. Moreover, the components should behave to composed in a matter that is small enough to be flexible. Simultaneously, the design pattern separation of concern is essential when building a distributed system. In chapter 3 we will go in-depth about how-to methods of separation of concern and software architecture for extensibility.

Being scalable Distributed systems aim to satisfy the demand for increasing demand from increased connectivity with various technologies, e.g. cloud services. Moreover, with the constant growth of users and applications, distributed systems include scaleable techniques to solve the challenges with scalability.

To structure scalability challenges, Neuman classified distributed computing scalability with tree dimensions [126]. These three dimensions include

1. size,
2. geographical and
3. administrative.

Neuman defines the dimensions: first, size refers to the number of the system's users, objects and services. Second, geographical refers to the distance between request and resources. Third, administrative refers to the number of different administrative organisations, e.g. research departments, using the system.

Each scalability dimension has its challenges. In the size dimension, the main challenges are dealing with an increase in requests and resources. From a centralised viewpoint, the request and resource increase must handle computational limits, data storage/transfer, and network issues. Second, the challenges lie in the component communication for the geographical dimension. For instance, many service architectures are based on synchronous communication. Therefore, communications need to be addressed with an extensive network of services. Third, the administrative

dimension deals with services being distributed over administrative departments. Finally, the main challenges lie in handling permission rights and security concerns. In general, the main challenges in scalability are handling a large number of requests and communicating and solving permissions to services.

To handle the challenges, we outline the main distributed scaling techniques. Generally, there are two approaches to handling scalability challenges: horizontal and vertical scaling. Vertical scaling also called scaling up, refers to the increase/improvement of resources. e.g. memory, CPU and network modules. Horizontal scaling - or scaling out - refers to deploying more machines in distributed systems. We illustrate the difference in horizontal and vertical scaling in figure 2.3.

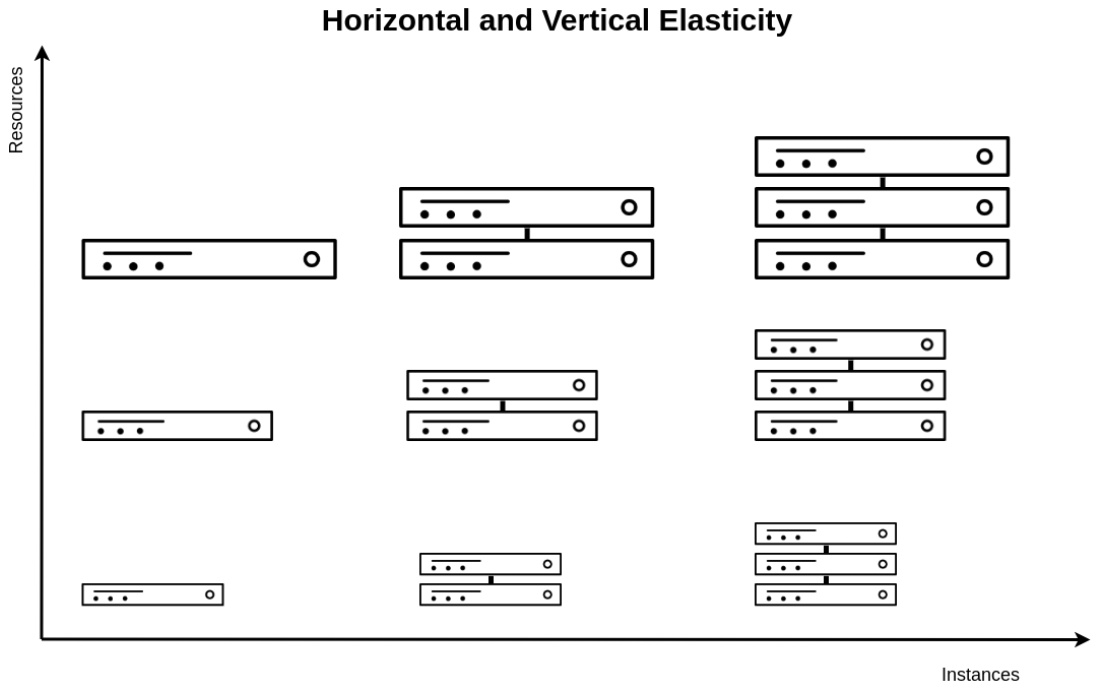


Figure 2.3: Horizontal and vertical scaling handle scalability challenges with different approaches. On the one side, vertical scaling aims to create more or improve resources, for instance, memory, CPU or network modules. On the other hand, horizontal scaling creates several instances of machines.

For horizontal scaling, there are three main technics that we can use: Hiding communication, distribution of work and replication. Hiding communication is a technique that solves the challenges in geographic scalability. This technic avoids

waiting for a response in a synchronous matter. Instead, this scaling technic proposes using asynchronous communication. For instance, rather than waiting for a response after a client requests a resource, the requesting application may execute other tasks. After the request is fulfilled, the client is notified of the ready result. Another technic is work distribution. This technic deals with software design and application decomposition with architectural software patterns. Finally replication technic creates several instances of parts of the system or components. The benefits of replication are an increase in availability and load balancing. We will detail these scaling technics in chapter 3.

Types of Distributed computing

One general scope of distributed computing is solving complex problems requiring high-performance computation. To execute high-performance computations, we can use two types of Distributed Computing: Cluster and Grid. These classes provide a design that helps execute a resource-intensive task. In cluster computing, most systems are arranged in a homogeneous collection of machines. These machines are similar in their hardware and software. Further, they are generally tightly coupled in a local area network. Contrary, distributed Grid computing generally spans a more comprehensive network of machines. This network may also span over a set of different administrative fields. Moreover, these machines can vary in their constitution with different hardware specifications and operating systems. Whereas grid computing uses loosely coupled heterogeneous machines, cluster computing uses a collection of tightly coupled homogenous machines.

Grid computing can be further subdivided into subtypes. These subclasses depend on the technical demand and application areas. In grid computing, one area of concern is the optimisation and efficiency of the system. Whereas another area is accessibility and opening services. One subclass of grid computing that focuses on accessibility and service availability is cloud computing. Cloud computing provides a pool of virtual resources for distributed grid computing. This service can be scaled to demand, as explained in the section above on design goals. The scaling possibilities will be explained in detail in chapter *3 Advanced Software Engineering in Bioinformatic: A Case Study of Design and Implementation* and *4 Enhancing Bioinformatics*

Analysis with Our Proposed System: Real-World Applications and Implications. In this section, we will expand on the general cloud computing architecture.

Cloud computing architecture is generally built on four layers: Hardware, infrastructure, platform, and application.

Hardware The basis of cloud computing is the physical hardware components, e.g. routers, processors, power and cooling systems. Usually, an end-user needs access to these components.

Infrastructure The infrastructure refers to the virtualisation and storage technical implementation. one cloud computing main task is to manage the virtual servers and storage devices.

Plattform Generally, the Platform layer is the interface that allows users to run an application in the Cloud. These interfaces are system-specific and may vary from chosen infrastructure implementation.

Application The application consists of the actual application. These applications can be customised and modified by the end user.

We illustrated these layers in figure 2.4.

Users have various options in choosing cloud providers with a different interfaces. These interfaces could include graphical, programming or command line. Genrally, these service paket are reffer to as follows:

1. Infrastructure-as-a-Service (IaaS) covering the hardware and infrastructure layer
2. Platform-as-a-Service (PaaS) covering the platform layer
3. Software-as-a-Service (SaaS) in which their applications are covered

Archtiechtural styles

Layered architecture One of the main objectives of general applications is to provide access to a database or any form of data. Moreover, we want to structure the process and components to request, process and represent data efficiently. For

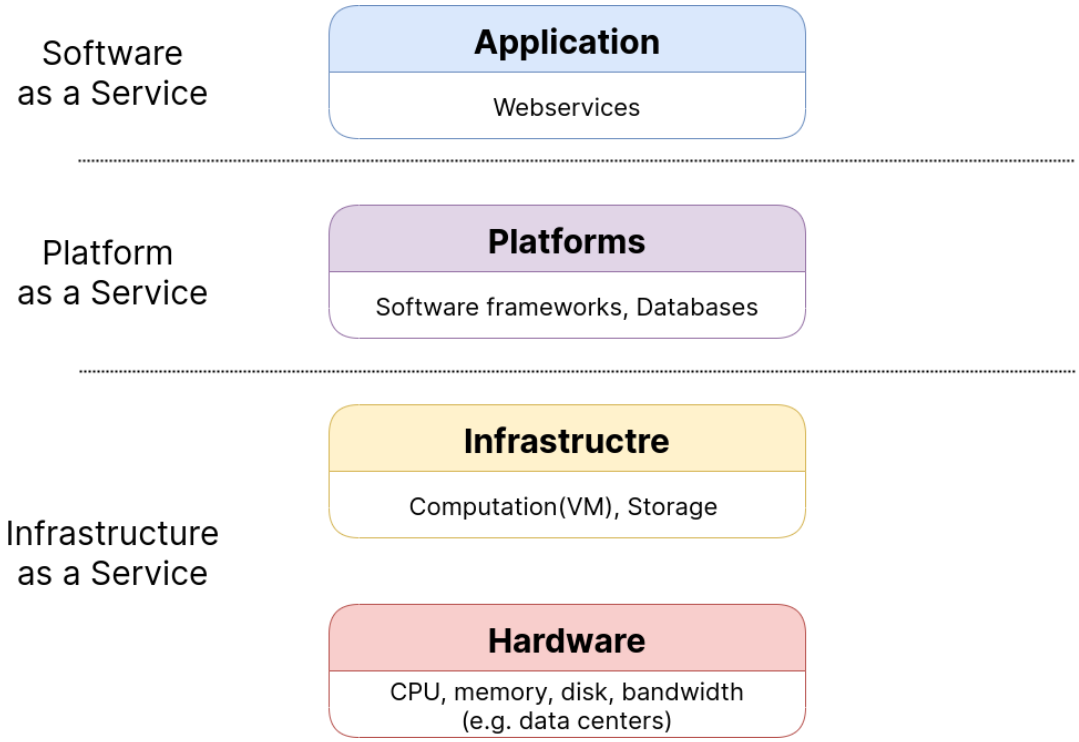


Figure 2.4: Cloud computing consists of four parts based on Zhang et al. [188]: Application, Platforms, Infrastructure and Hardware. These four parts can then be grouped into three services customers can subscribe to software-, platform- or Infrastructure-as-a-Service.

this task, we can use an architectural pattern that separates concerns and provides an interface to communicate. This architectural pattern is referred to as layered architecture. Layered architecture general functions by a component L_i making a request /call calls to a lower level L_j with $(i < j)$. This request also referred to as a downcall. For example, for L_j calling L_i, we refer to this call as upcall. Further, the communication from a higher to a lower layer is defined with a predefined interface. All in all, layered architecture provides an architectural pattern to solve data access possibilities. We illustrated layer architecture in figure 2.5.

Specifically when we use a logical layering for an applications there is a frequently used approach. This approach generally consist of three-part:

- application-interface level,
- processing level and

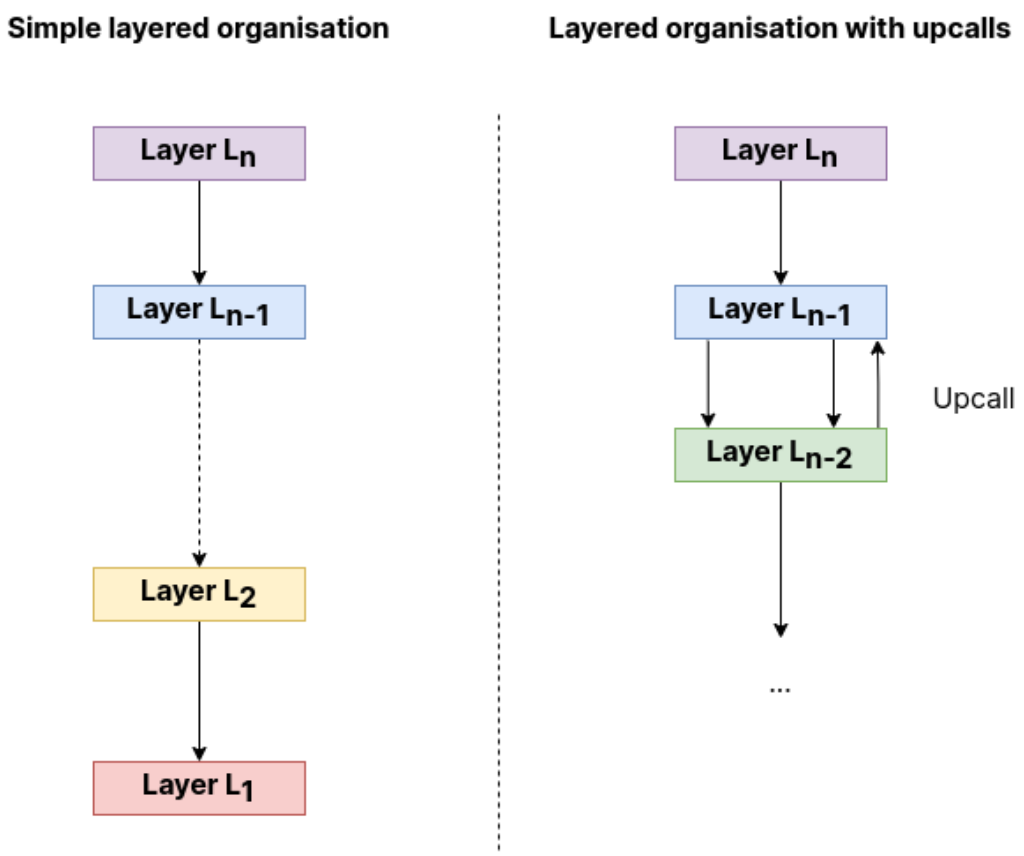


Figure 2.5: Based on Krakowiak, the layered architecture allows the separation of components [94]. In a simple layered organisation, only down calls are allowed. Another possibility is to allow back-and-forth communication is allowing upcalls.

- data level

The application-interface level concerns the interaction with a user or other application. The data level operates on the database or raw data on a file system. The processing part contains the core functionalities. The core functionalities most often include logic and methods. Whereas the application interface and data level are very similar in many applications, the middle layer can have various characteristics. Researchers and developers use and modify these three layering levels for their demand depending on the characteristical context. This three-level layering is also referred to as three-tier architecture.

Object-based and service-oriented architectures When designing applications, one frequent challenge is that resources may be scattered around several administrative areas. Moreover, software dependencies might conflict when using many different tools and applications. To solve these challenges, we can use an object-based or service-oriented architecture. These architectural patterns encapsulate the object or services into independent environments. Moreover, each object or service clearly defines these environments' states, methods, and interfaces. We illustrated this architecture in figure 2.6. All in all, object-based and service-oriented architecture encapsulates objects and services to connect scattered administrative applications.

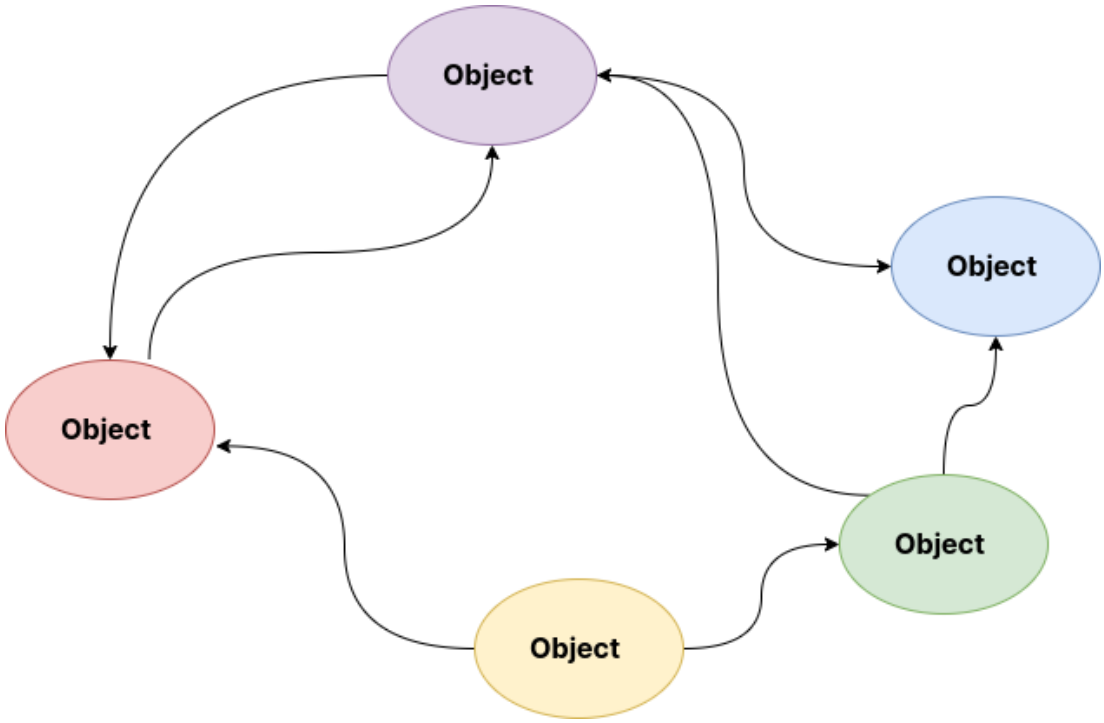


Figure 2.6: Each component is encapsulated in an independent environment in an object- or service-based architecture. These components communicate with each other via predefined interfaces.

We outline the general function of Object-based and service-oriented architectures to describe how object communication and encapsulation is designed. Generally, object-based architecture defines each object as a modular unit with well-defined interfaces - similar to components. These modular units communicate via a developer-designed communication mechanism. In this design, the object implemen-

tation is separated from the interface. This separation allows the interchangeability and independence of objects. The concept of interchangeability and independence is the basis for service-oriented architecture. A service is a self-contained entity. In a service-oriented architecture, an application is composed of a set of services. We will detail the service-oriented architecture and system decomposition in chapter 3. Altogether, object-based as well as service-oriented architecture encapsulates state, method and interface into one object or service using a developer-designed communication method to connect services.

Resource-based architecture A service-oriented architecture requires a technical implementation to handle its decomposition. The decomposition of an application grows in complexity with an increasing number of services - mainly web services. This increasing service complexity can be managed with design patterns from Resource-based architecture. Resource-based architecture generally uses a restful interface to connect its core entities. These core entities are referred to as resources. In scope of Resource-based architecture defines an encapsulated entity, e.g. files, data or requests, obfuscating its implementation. From a general perspective, resource-based architecture can be viewed as a subclass of service-oriented architecture since it uses encapsulation similar to services. In this matter, resource-based architecture further specifies the technical implementation with a REST-ful interface. Resource-based architecture provides a technical design that can organise many services to deal with the high increase of services.

Richardson et al. summarised resource-based architecture in four core concepts:

- definition of resource,
- naming and addressing resources,
- representation of resources and
- linking resources together.

Generally, a resource is defined as a piece of data that can be stored on a computer. This data may include a file, database request results, or an algorithm's output. To access these resources, they require a name and address. This naming and addressing is referred to as Uniform Resource Identifier (URI). The URI exposes data for any

client to consume the data. The resource requires a pattern or design to send the data for further data processing. This design depends on the developer's choice. The design can range from JSON format to plain text. Finally, for a resource-based architecture, resources can be linked together so that a resource might reference to list of other resources. Richardson et al. provided technical core resource-based architecture concepts that guide developers in building applications.

Middleware organization

Similar to general application development, distributed computing faces two common challenges. First, legacy and new tools are integrated with different or proprietary interfaces. Second, it can be challenging to change the data flow after establishing an infrastructure. Distributed computing solves this issue by creating a software layer between the operating system and the applications. This software layer is also referred to as middleware. Generally, middleware benefit is the separation of concern from the technical task. These tasks may include communication, transaction, service composition and reliability. In addition, middleware provides a software layer to handle the new application and data flow.

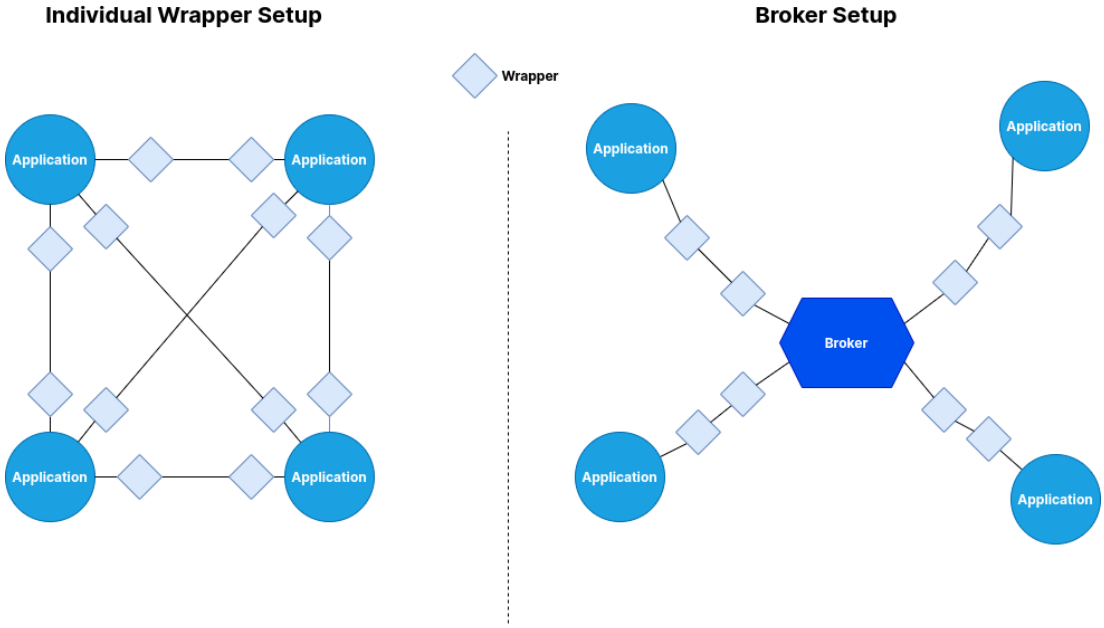


Figure 2.7: Middleware uses the wrapper pattern to organise the system

Middleware uses several design patterns To provide an infrastructure to organise distributed systems. Two of these patterns are Wrappers and Interceptors. A wrapper pattern solves the challenge of integrating new and legacy tools to comply with an interface expected by a client. The component that manages conversion between client and server/application is also referred to as a wrapper. A centralised technic can optimise the number of wrappers in a distributed system. Generally, a system with N components must have fully bidirectional communication $N * (N - 1)$ wrappers. This matter can be improved by using a centralised component, also referred to as a broker, to handle communication. Then the number of wrappers is $2N$. We illustrated wrappers as graphs in figure 2.7. Apart from wrappers, interceptors further organise a system's middleware. An interceptor enhances existing services with new capabilities. In other words, an interceptor can change the data flow in a distributed system. Interceptors are interposition objects that can intercept calls and add additional processing or analysis. We illustrated the data flow adapted with an interceptor in figure 2.8. The wrapper and interceptor patterns are key in integrating new and legacy tools and adapting the data flow.

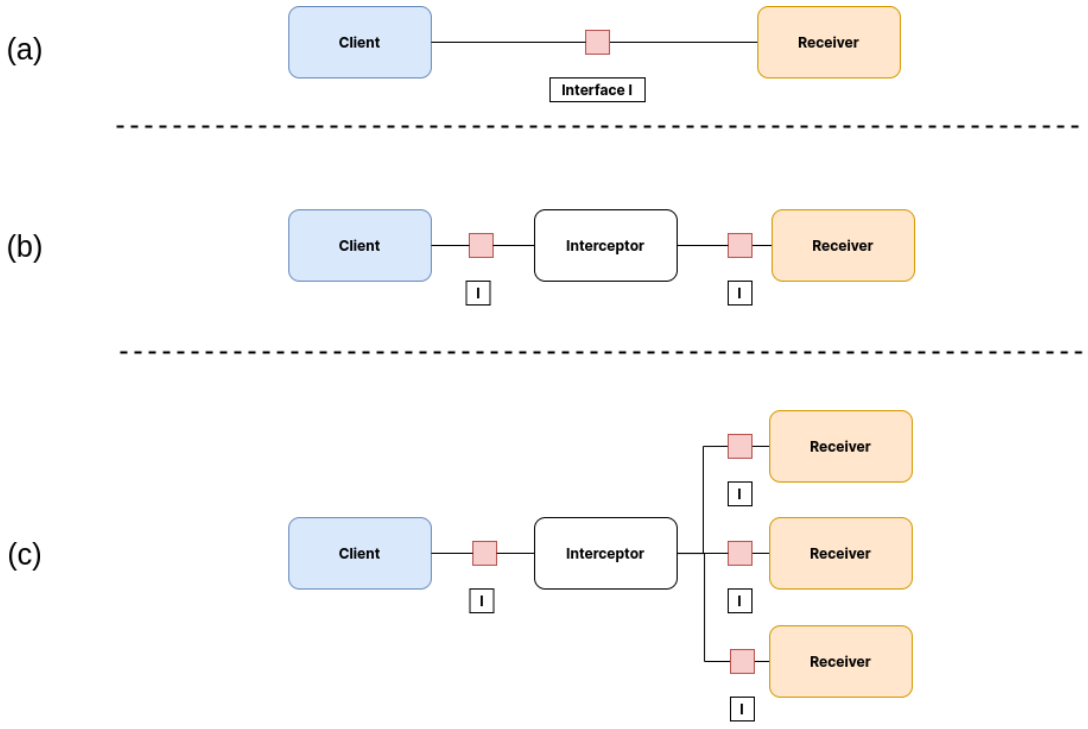


Figure 2.8: with interceptor an the system can change and modify the flow of data

2.2.3 Containerisation

This section outlines Containerisation and provides a general overview of the field. Concretely, this section gives contextualises Containerisation is part of virtualisation. Further, we describe implementations of containerisation. Finally, we discuss technologies to manage containerisation.

Overview

Containerisation is form of virtualisation. Virtualisation refers to the replication of a hardware or software object by a similar object of the same type using an abstraction layer. This abstraction allows virtual (i.e. non-physical) devices or services such as emulated hardware, operating systems, data storage or network resources to be created. Virtualisation can be classified commonly using two technology patterns: hypervisor-based or container-based. For hypervisor-based virtualisation, the virtualisation requires — as the same suggests — a hypervisor. A hypervisor is a piece of software that supplies a guest operating system as a virtual operating system. Another term customarily used for virtual operating systems is virtual machines. We differentiate the virtual machine from the machine it is running on. We use the term guest for the virtual machine and host for the machine it runs on. Further, the hypervisor creates virtual resources such as processing, memory and storage. In contrast to hypervisor-based virtualisation, container-based virtualisation utilises system isolating mechanics to encapsulate an environment. These mechanics rely on Linux core technologies, e.g. namespaces and control groups. Therefore, encapsulated containers are isolated, independent processes/applications from a technical standpoint. The benefits of containers are that they are flexible, scalable and resource-efficient. Container-based virtualisation eliminates resource handling for managing OS kernels, libraries, and binaries. These software components are required to run workloads or applications in a virtual machine in hypervisor-based virtualisation. Figure 2.9 illustrates the application deployment using a hypervisor and container-based architectures. Altogether, the benefits of containerised virtualisation become clear compared to hypervisor-based virtualisation.

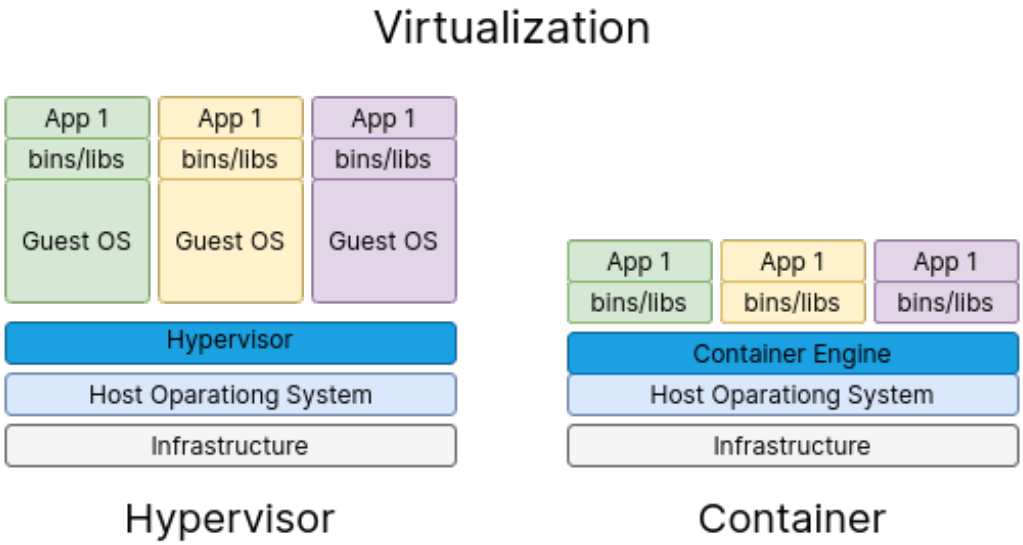


Figure 2.9: The compirisant between container and hypervisor virtualization. since there is no OS in each instance in a container, it uses fewer resources the hypervisor-based deployment requires different operating systems and adds an extra layer of virtualisation compared to containerisation.

Container-based Implementations

Many container-based virtualisation implementations vary in their approach to handling the containers. We outline three commonly used implementations: Docker and Linux. Containersand and OpenVZ.

Docker Docker simplifies the deployment of applications because containers containing all the necessary packages can be easily transported and installed as files. In addition, containers ensure the separation and management of resources used on a computer. According to the developers, this includes code, runtime module, system tools, system libraries - everything installed on a computer. Docker is based on Linux techniques such as Cgroups and Namespaces to realise containers [10]. We will expand on Docker in chapter 3.

LinuX Containers (LXC) LXC (Linux Containers) is an operating system-level virtualisation method that allows multiple Linux systems run in isolation from each other on a single host. Unlike other systems, LXC does not realise its virtualisation

using virtual machines. Instead, LXC creates a virtual environment that has its processes but shares the host system's kernel. LXC consists of a program library, various Python, Lua, and Go APIs, container templates, and tools for controlling the containers. Besides kernel namespaces, it uses other functions of the Linux kernel, such as SELinux and groups [12].

OpenVZ OpenVZ (Open Virtualization) is software for Linux for virtualisation of the operating system. OpenVZ creates multiple isolated containers for operating systems. All processes of these operating systems are processed in a single kernel. The operating systems in the containers are nevertheless largely independent of each other. For example, they can be shut down independently, and each has its root account [9].

others Other implementations include Linux-VServer [4] and Singularity [13]

In this thesis, we focus on Docker since Leipzig et al.proposed Docker as a containerisation technology to create pipelines [106]. However, we added other Linux-based virtualisation possibilities in section *A.2.3 Linux containerisation*. We will go in dept over the Docker technology in chapter *3 Advanced Software Engineering in Bioinformatic: A Case Study of Design and Implementation*.

Container Management and Orchetraction

One central concern in containerised virtualisation is managing the growing number of containers. The increase in amount proliferates the challenges in deployment, communication and data storage. A commonly used solution is a software layer that handles these concerns. This software layer is also referred to as container orchestration. The container orchestration provides an architecture that handles container-based virtualisation. Further, in most implementations, these infrastructures can span over several hosts managing a wide range of tools to handle container-based virtualisation. Overall, Container orchestration is a software layer that handles extensive container-based virtualisation.

Similar to container-based virtualisation, there is a variety of container-based orchestration implementations. However, since we focus on Docker-based containerisa-

tion, we outline three primary orchestration tools: Swarm, Kubernetes and Rancher.

Docker Swarm Docker Swarm is a clustering and scheduling tool for Docker containers. With Docker Swarm, Docker clusters can be created and managed like a single virtual system. Clustering is an essential feature of container technology because it allows the creation of cooperative systems groups to prevent node failure through redundancy. Clustering also allows administrators and developers to add or remove container iterations as computing requirements change. In addition, Docker Swarm uses scheduling to ensure that there is always enough capacity for distributed containers. The Docker tool assigns containers to underlying nodes and optimises resources by automatically scheduling container workloads, ensuring they always run on the optimal host. Docker Swarm thus provides primary workload balancing for containers, and sufficient resource coverage [7].

Kubernetes Kubernetes is an open-source system for automating the deployment, scaling and management of container applications, initially designed by Google. It aims to provide a *platform for automating application container provisioning, scaling and maintenance on distributed hosts* [11]. It supports a range of container tools, including Docker.

Kubernetes orchestrates so-called *pods* as the smallest deployable unit. Pods contain one or more containers, sharing a container runtime and allocated resources. Pods are deployed and executed on *nodes* (physical or virtual machines in a cluster).

The cluster with its nodes is controlled by a dedicated machine and the *Kubernetes Control Plane*, which communicates with the individual nodes via the *Kubelets* running in them. The Kubernetes Control Plane runs an instance of etcd, the central key-value database for all information essential for managing the cluster, as well as the automated controller processes and a scheduler that assigns newly created pods to a node.

The controllers monitor and control the cluster and its components. They can, for example, replace failed nodes with identical nodes.

Rancher Rancher is an open-source cluster management tool for Docker containers. From an abstract perspective, Rancher lies on top of other orchestrators such

as Kubernetes, Docker Swarm, Apache Mesos. This arrangement facilitates running other container clusters [119].

In this section, we have briefly presented some of the container orchestrators. However, there are other orchestrators, such as OpenShift, Cattle.

2.3 Related work: A literature review of the previous studies and their impact on the current research

Extensive data management and the growing demand for biomedical analysis have led to the development of specialized frameworks that provide different solutions with varying focuses. For one set of systems, key features include a specialized field of application, an intuitive user interface, and a secure software setup, such as GATK. Other methods focus on other aspects, such as containerized software environments like Argo. While some structures focus on specific biomedical aspects, others are geared for general use across several industries. These two sets of frameworks have different audiences. The first set serves highly specialized biomedical researchers with a focused application, such as genomic sequence analysis. For these frameworks, the possibility of expansion can be tricky since they focus on specific investigations. The second set serves software engineers who build business data-driven systems. Engineers commonly deploy these systems on cloud providers with existing architectures. Admittedly, one could deploy this software architecture on-premise, but it would require additional setup overhead.

Moreover, the topic of Big Data and Cloud Computing has experienced tremendous development in the industry and the academic field. Many industries and applications have moved to cloud computing for scalability and flexibility by dynamically allocating resources to meet demand [2]. This development allows, for example, to scale of processing-intensive computational tasks accordingly and reduces computing times. Platforms offered by large companies, such as Google and Amazon, provide an infrastructure that offers solutions for software deployment. In 2019, already 78% of companies used cloud computing in some way to run their business [66] with a

predicted annual growth rate of 12.6% by 2022 [42]. In biomedical data analysis, many frameworks try to incorporate container technology, but there are yet to be ready-made solutions for containerisation with distributed computing. For this reason, there are several solutions to deploy cloud architectures. In biomedical data analysis, several frameworks incorporate containerisation, such as snakemake and Pachyderm. From an industry standpoint, there are many cloud containerisation frameworks with various degrees of requirements. One of the core issues is the setup, as mentioned before.

This section aims to provide a comprehensive overview of the state-of-the-art in the related fields of bioinformatics data analysis, decomposing architecture into microservices, and data management frameworks. We begin by delving into the various software systems available for bioinformatics data analysis, highlighting their unique features and capabilities. Subsequently, we delve into the architectural pattern of microservices, discussing the potential benefits and challenges associated with their implementation in bioinformatics systems. Finally, we focus on data management frameworks, exploring their significance in effectively organizing, storing, and managing large and complex bioinformatics datasets. Through this section, we aim to provide a thorough understanding of the current advancements in these related fields and their potential implications for the field of bioinformatics.

2.3.1 Software systems for bioinformatics data analysis

Data analysis is a crucial aspect of bioinformatics research, and a wide range of frameworks and platforms have been developed to facilitate this analysis. These solutions vary in focus and features, from specialized solutions designed for specific types of data and algorithms to more general software workflow systems that can be applied across various disciplines. Here, we provide a sample of the diverse bioinformatics data analysis frameworks and general software workflow frameworks. By outlining and understanding these frameworks, we can gain a comprehensive overview of the options available for extensive data set analysis in bioinformatics. Furthermore, by providing this overview, we aim to identify areas of opportunity for further study in

bioinformatics data analysis.

Bioinformatics frameworks

Several frameworks have a specific focus, such as the Genome Analysis Toolkit (GATK) proposed by McKenna et al. GATK is a programming framework that simplifies the development of efficient and reliable variant calling tools for next-generation DNA sequencing using MapReduce. GATK enables fast and easy tool creation optimized for accuracy, stability, and efficiency and can be parallelized for distributed and shared memory systems [120]. Another framework is Gesall, a big data platform for genome analysis pipelines based on Wrapper Technology. Gesall supports existing genomic data analysis programs without requiring rewriting. In addition, the framework includes the Genome Data Parallel Toolkit for "wrapping" programs and enabling evaluation of big data technology for genomics with super-linear and sublinear speedup and potential differences in results between parallel and serial programs [149]. Another framework is META-pipe, a new pipeline for marine metagenomics analysis that offers preprocessing, assembly, taxonomic classification, and functional analysis. It is integrated with existing biological analysis frameworks, distributed storage, and Supercomputer computation, with a web service providing identity provider services, Galaxy workflows, and interactive data visualizations. It has been evaluated for scalability and performance [148]. Another example is Galaxy, a collaborative global project that provides web-based tools and resources for analyzing large biomedical datasets and has seen growth in code contributions, tools, users, and training materials, including enhanced user interfaces and comprehensive frameworks. In addition, new public servers and community resources [87].

Bioinformatics data analysis frameworks and solutions often focus on a specific application, such as metagenomics or variant calling. While these systems may be effective for their intended purpose, their specialized design can limit their expandability and flexibility. Further, researchers may be locked into using a single framework when choosing one solution. This situation can be problematic as researchers may need to use a variety of tools and approaches to analyze their data effectively but may need to be improved by the capabilities of the system they are using. Expanding the functionality of these systems may require custom program-

ming, which might conflict with the intended design of the system. The inability to quickly adapt a framework to incorporate new tools and methods can hinder research progress and limit the ability to thoroughly exploit the potential of the data being analyzed. Therefore, researchers must carefully consider the expandability and flexibility of the systems they choose to use to ensure that they can effectively and efficiently analyze their data and make the most of the insights it can provide.

General workflow framework

Besides highly focused tools, biomedical systems and general-purpose software workflow engines cater to the broader technology industry and share several similarities in analyzing biomedical data. For instance, the biomedical workflow management system Snakemake is a popular workflow management system. Researchers use Snakemake to ensure data analysis reproducibility, adaptability, and transparency. In addition, the tool provides an ergonomic, unified representation of all steps involved, from raw data processing to final result exploration and plotting [93]. Finally, on the more general-purpose side, there is GNU make. GNU Make is a popular tool used to manage the construction of complex software systems and data analyses, allowing for the automation of repetitive tasks and the reproduction of results. In addition, it enables the efficient and transparent execution of processes.

Moreover, it makes a valuable tool for achieving reproducibility, adaptability, and transparency in research [161]. Finally, Navarro suggested a text mining system with promising results: Argo [24]. Argo is a text-mining workbench that allows users to build custom text-mining solutions by integrating various elementary components into processing workflows. It enables domain experts to curate information of interest through a graphical annotation interface using the workflows' automatically generated output. Toil is an open-source workflow software that allows users to efficiently process large-scale genomic data sets in cloud or high-performance computing environments. It includes a complete set of features, including fault tolerance, cloud support, and HPC support, making it capable of efficiently processing petabyte-sized data sets and producing results faster and for less cost across diverse environments [176]. Arvados is an open-source platform that allows users to manage, process, and share large genomic and biomedical data sets. It is designed to handle

big data files such as genomes, tumour/normal pairs, microbiomes, and other data. It can handle data sets ranging from tens of terabytes to petabytes. In addition, Arvados provides capabilities for bioinformaticians and computational biologists to run pipelines and applications on top of it [8].

Biomedical data analysis often requires the implementation of complex workflow systems. While general workflow engines can be used to accomplish this task, the setup and maintenance of these systems often require a strong background in software engineering. In addition, the underlying system infrastructure also has a range of requirements, including installing a job scheduler (e.g. Slurm or Portable Batch System) or container orchestration tool, such as Kubernetes [173].

Certain drawbacks may accompany business-oriented cloud approaches for biomedical data analysis. For example, scientists may need to store sensitive data on external servers. On the other hand, researchers may face challenges in setting up and configuring the necessary infrastructure, including the manual setup of components such as the load balancer and control plane [5]. In addition, setting up a network layer over the cluster can be a complex task [25], and the setup of a Kubernetes cluster may require a dedicated team. Even small companies delegate the maintenance of a Kubernetes cluster to a separate department.

Current frameworks for biomedical data analysis could be divided into two categories: those that are highly specialized but lack expandability and containerisation, such as Arvados [8], and those that offer containerisation but require significant overhead for setup, configuration, and maintenance, such as Argo [24]. We summarize critical features in table 2.1. Finally, scientists may encounter difficulties finding a framework that meets their needs and aligns with their expertise.

2.3.2 Decomposing architecture into microservices

Researchers in the field of bioinformatics have recognized the crucial role that software systems play in the analysis and interpretation of biological data. These systems often require the use of complex algorithms and data processing capabilities. However, the development and maintenance of these systems can be challenging due to their size and complexity. As a result, many bioinformatics software systems adopt software engineering approaches such as microservice decomposition to address these

	GATK [120]	snakemake [93]	META-Pipe [148]	Arvados [8]	Presented Framework
Creation	2010	2012	2016	2018	2020
Containerized SOA system	no	no	no	no	yes
Set up requirement	none	Kubernetes	none	Docker*/ Kubernetes	Docker
Virtualisation	no	Docker	no	Docker	Docker
Workload Manager	Google Cloud	Kubernetes/Google Cloud	none	Kubernets/SLURM	Docker Swarm
Web API	RESTful	no	no	Yes	RESTful
Execution environment	cloud	cloud	HPC Cluster	cloud/ HPC	cloud
Task creation when deployed in cloud	command line	command line	not available	command line/GUI	curl/GUI
Simplified task creation	none	none	none	none	GUI

Table 2.1: Medical software platforms for data analysis with their software components as well as suitability for interactive visualisation or processing. *: The Docker deployment in Arvados is mainly intented for testing, development and demonstration

challenges.

Microservice decomposition involves breaking down the system into more minor, independent services that can be developed, tested and deployed independently. This approach enables greater manageability and maintainability of the system. As well as this approach improved scalability and adaptability to meet changing needs and requirements. Additionally, microservice decomposition can facilitate collaboration and allow the use of diverse technologies and programming languages within a single system.

A vast list of research on microservice decomposition proposes several approaches. These approaches include supporting developers in decomposing their systems into optimal sets of microservices. For example, Abbott and Fischer proposed a decomposition approach based on the "scalability cube," which splits an application into smaller components for increased scalability [14]. Richardson also mentioned this approach in his four decomposition strategies: decomposing by business capability, decomposing by domain-driven design subdomain, decomposing by a verb or use cases, and decomposing by nouns or resources [147]. Kecskemeti et al. proposed a

decomposition approach based on container optimization to increase the elasticity and flexibility of large-scale applications [90]. Zimmerman et al. proposed to move towards a microservices-based architecture or to deliver separate microservices. They suggest splitting a development team into smaller groups responsible for limited sets of microservices [191]. Vresk et al. defined an Internet of Things (IoT) concept and platform based on the orchestration of different IoT components and recommend combining verb-based, and noun-based decomposition approaches [177]. Gysel et al. proposed a clustering algorithm approach based on 16 coupling criteria. They introduced the concept of coupling criteria cards, which was evaluated by integrating two existing graph clustering algorithms, a combination of action research and case study investigations, and load tests [78]. Chen et al. proposed a data-driven microservices-oriented decomposition approach based on data flow diagrams from business logic [36]. Alwis et al. proposed a heuristic for slicing a monolithic system into microservices based on object subtypes and functional splitting [48]. Lastly, Taibi and Systä proposed a 6-step framework. The approach reduces subjectivity in the decomposition process of monolithic systems into microservices [168]. The framework provides options and evaluation measures identified through a process-mining tool on log traces. The framework was able to identify previously undiscovered issues and suitable decomposition options. Moreover, it proved its validation in an industrial project. The framework helps improve the quality and objectivity of decomposition in any monolithic system.

Selecting the appropriate tool or approach for designing and decomposing a system into microservices is complex and challenging. Despite the availability of various options, each with its unique features and capabilities, determining the most suitable one for a specific project can take time and effort. Factors such as the specific needs and requirements of the project, the technical skills and expertise of the development team, and the available resources must all be considered when making this decision. Ultimately, the goal is to choose a solution that best meets the project's needs and supports the development team in achieving its objectives.

2.3.3 Data management frameworks

Bioinformatics involves managing and analysing vast amounts of biological data, including genomic sequences, protein expressions, and molecular interactions. The effective handling of such data is crucial for advancing life sciences research and supporting decision-making in areas such as drug development and personalised medicine. Specialised data management systems have been developed to address the challenges of managing and analysing biological data. These systems often incorporate features such as efficient storage and retrieval of large datasets, data integration and interoperability support, and data visualisation and analysis tools. In this section, we will explore the various general software approaches and technologies used to design and implement data management systems for bioinformatics.

The concept of complex data and its implications for data warehousing is addressed in the work of Darmont et al. [45]. They propose that the complexity of data can be characterised in various ways and offer an expanded definition of a data warehouse that considers complex data. In addition, Waas [179] suggests that the traditional order of transformation and loading in data warehousing can be altered, with raw data being loaded into the warehouse and transformed for later analysis. These findings have important implications for the design and management of data warehouses, mainly when dealing with complex data.

One commonly used programming model for handling large amounts of data is the MapReduce model. MapReduce is a programming model and an associated implementation for efficient processing and generating large data sets [49]. It involves the specification of a map function that processes a key/value pair to generate intermediate key/value pairs and a reduce function that merges all intermediate values associated with the same intermediate key. The resulting programs can be automatically parallelised and executed on a cluster of commodity machines, with the runtime system managing the details of partitioning the input data, scheduling the program's execution, handling machine failures, and facilitating inter-machine communication. This allows for using distributed systems without requiring expertise in parallel and distributed programming, enabling the efficient handling of large amounts of data.

Several frameworks implement the MapReduce model. One popular framework

is the Hadoop Distributed File System (HDFS) [157, 181]. It has been designed to reliably store and stream large data sets at high bandwidth to user applications. It utilises a distributed architecture across many servers, each with directly attached storage, allowing it to scale economically as demand grows. Another popular framework is Apache Spark, a fast and general-purpose cluster computing system [1]. Spark extends the programming model of MapReduce by improving speed and general processing while extending the programming API to allow more flexible chains of map functions. It also covers a broader range of workloads by including batch applications and iterative algorithms. Apache Flink is another open-source system for processing streaming and batch data that allows a wide range of applications. Flink includes real-time analytics, continuous data pipelines, historical data processing, and iterative algorithms, to be expressed and executed as fault-tolerant dataflows through a single unified execution model [32].

Although MapReduce frameworks such as Spark and Hadoop offer robust scalability and flexibility, they have certain limitations regarding deployment in a heterogeneous bioinformatics environment. These frameworks are typically deployed using pre-built images, such as Amazon Machine Images (AMIs), which contain all necessary software and configurations. However, this can limit the flexibility of deployment in a bioinformatics environment where specific software or configurations may be required. Additionally, these frameworks may need to be optimised for handling certain types of complex biological data, such as high-throughput sequencing data. Therefore, it is crucial to consider a bioinformatics project's specific needs and requirements when selecting a data management framework.

In conclusion, recent years have seen significant advancements in bioinformatics data analysis, decomposing architecture into microservices, and data management frameworks. Software systems for bioinformatics data analysis have become more sophisticated and efficient, providing a wide range of data analysis and visualisation capabilities. The microservices architectural pattern has emerged as a promising approach for building scalable and maintainable bioinformatics systems. Furthermore, data management frameworks have become increasingly crucial for effectively organising, storing, and managing large and complex bioinformatics datasets. This section has provided a comprehensive overview of the state-of-the-art in these re-

lated fields and highlighted their potential implications for bioinformatics. However, despite these advancements, open issues and research challenges still need to be addressed to improve bioinformatics systems' capabilities further. In the next section, we will delve into these open issues and research challenges, providing insight into the current limitations of bioinformatics systems and identifying potential areas for future research.

2.4 Open issues and research challenges in biomedical data analysis

Despite the diverse research and development in containerised distributed computing in biomedical data analysis, there are still open issues concerning system design, implementation, and integrations [46]. This section outlines the open issues and challenges grouped by the concern: software, data, and biomedical. There is considerable overlap in these areas since they interactively influence each other. For instance, software architecture influences how workflows can be executed, while the dependencies of a workflow influence the required software architecture. Nevertheless, partitioning these concerns helps give the context and orientation of the challenges. This section provides the context and impact of open challenges in biomedical data analysis.

2.4.1 Software engineering for biomedical data analysis

Biomedical data-driven analysis requires robust and adjusted software systems to execute analysis reliably and repeatably. Therefore, software systems architecture relies on the infrastructure for all further analysis. However, software architecture is regarded as a prerequisite since biomedical software development is still in its early stages [46]. Similar to other data-driven fields, biomedical data-driven research focuses on algorithmic advances. These advances depend on robust computing systems. Therefore, this section will outline open challenges in the field of data-driven biomedical analysis.

Customised software architecture When designing an application, one main concern is software architecture. In a study in 2013, Avci et al. outlined the vast spectrum of software architectures for data analysis [19]. We infer from the survey that a fitting architecture for data analysis involves much consideration. One of the critical concerns is requirement management. Requirements may range from stakeholders to technical functionality. Chen et al. outlined the implication of requirements on the software architecture and vice versa [35]. Therefore, one open research question remains about what architecture to use for our given use case of biomedical data analysis.

Software maintenance Dependency management remains a challenge in software deployment. Many developers use a simple deployment and forget about the software dependencies. With the fast new development, it is imperative to consider managing software dependencies. Conflicting dependencies can have adverse effects on software functionality and operating systems. All in all, package and dependency management is an essential part of deployment and software management.

Moreover, software maintenance needs to be addressed in software development. One of the critical aspects of missed points is technical debt. Technical debt refers to the additional work that needs to be done after a software installation for not considering what has to be done. Moreover, software needs to be updated and maintained; otherwise, it will be chaotic. Refer to the Deutsche Bahn example when they used windows XP, NPM big and log4j. Keeping software up to date and maintaining it to close security issues is imperative.

Using containerisation with distributed computing With the advancement in cloud services, many developers use containerised virtualisation to decompose services. Service decomposition benefits from independent environments. These independent environments allow benefits for scalability, reliability, security, high maintainability, testability, and deployability. Therefore, containerisation is a beneficial approach to service decompositions.

Nevertheless, when using containerisation, several aspects have to be considered by developers. Many developers turn their attention to ready-made solutions or frameworks such as different services, e.g. platforms-, infrastructure- or software-

as-a-service. Generally, it is recommended to separate software infrastructure and concerns so that developers can focus on features. However, relying on other ready-made frameworks or infrastructure entails two main drawbacks: privacy and setup. Concerning the setup, many ready-made frameworks have (to consider many requirements) software requirements. These requirements need a considerable skill set to set up and maintain a production-grade environment. Concerning privacy: when using third-party cloud services, the provider stores the data in their facilities. This data storage arrangement conflicts with sensitive data — especially medical data. Therefore, using the third-party provider option is inapplicable. Generally, privacy aspects and setup have to be considered when using cloud containerisation methods.

Handling legacy and modern tools One of the challenges in software management for biomedical data analysis is facilitating consistent environments for experiments and simulations. One key issue is that clients' operation systems application might influence the experiment outcomes from a software management perspective. Moreover, the difference in operating systems brings an additional step of complexity in ensuring consistent environments. Generally, software infrastructure is essential in allowing consistent biomedical data analysis.

In addition, one general software management issue is the package management remains a key concern in running biomedical data analysis. Biomedical data analysis utilises a vast array of tools and methods. This spectrum may vary from a legacy application to newly published methods. One of the main concerns with using all these tools is handling the dependencies. Conflicting dependencies can affect experiments by falsifying the results or even bringing the systems into an unresolvable state where booting the system is impossible. Therefore, dependency management takes on a key role in building a reliable software system for biomedical data analysis.

For biomedical analysis, one essential requirement concerning software development is integrating various tools. These tools may vary from legacy to novel, experimental tools. In addition, ensuring a consistent and repeatable software environmental condition is a complex task. Therefore, providing a custom system design for biomedical analysis remains an open challenge.

2.4.2 Data management in biomedical research

At the centre of biomedical data analysis is — as the name suggests — data. The data-driven analysis aims to create value from data as one of its primary goals. Value can be understood as finding relationships in data or gaining further insights. Data-driven analysis has experienced many new developments with various approaches and concerns. These approaches and concerns have intertwined goals.

In this section, we propose a model for structuring data concerns across three key areas: data handling, analytics, and result extraction. As shown in Figure 2.10, these areas are interrelated and overlapping, with individual challenges often lying at the intersection of multiple subfields. Despite these complexities, we provide a general mental framework for organizing data analysis and address the challenges associated with data handling, analytics, and result extraction. In particular, we focus on the open challenges in handling data, as effective data management is a critical precursor to all further analysis. By addressing these challenges, we aim to provide a foundation for more effective data-driven approaches in biomedicine.

Data organization is a crucial starting point for any data analysis, with several key concerns to be addressed to generate value from the data. These concerns include unstructured and scattered data, data maintenance, large data sets, and system failure recovery. Therefore, data handling must address a broad spectrum of concerns as a foundational step towards value creation.

Of particular interest is the combination of genomics and clinical health data, which has the potential to enable predictive and personalized medicine [46]. However, the use of clinical health data introduces additional challenges, such as data transfer and patient privacy concerns [41], which must be carefully considered in the workflow of using this data, especially in a clinical routine setting.

Unstructured and Scattered Data In this paragraph, we will outline the problems with unstructured and scattered data. First, we define what is meant by unstructured data and outline the open challenges. Afterwards, we will go over the scattered data and the open issues.

Before going over challenges concerning unstructured data, we outline the general concept around the term. Data can be grouped into three classes from a software

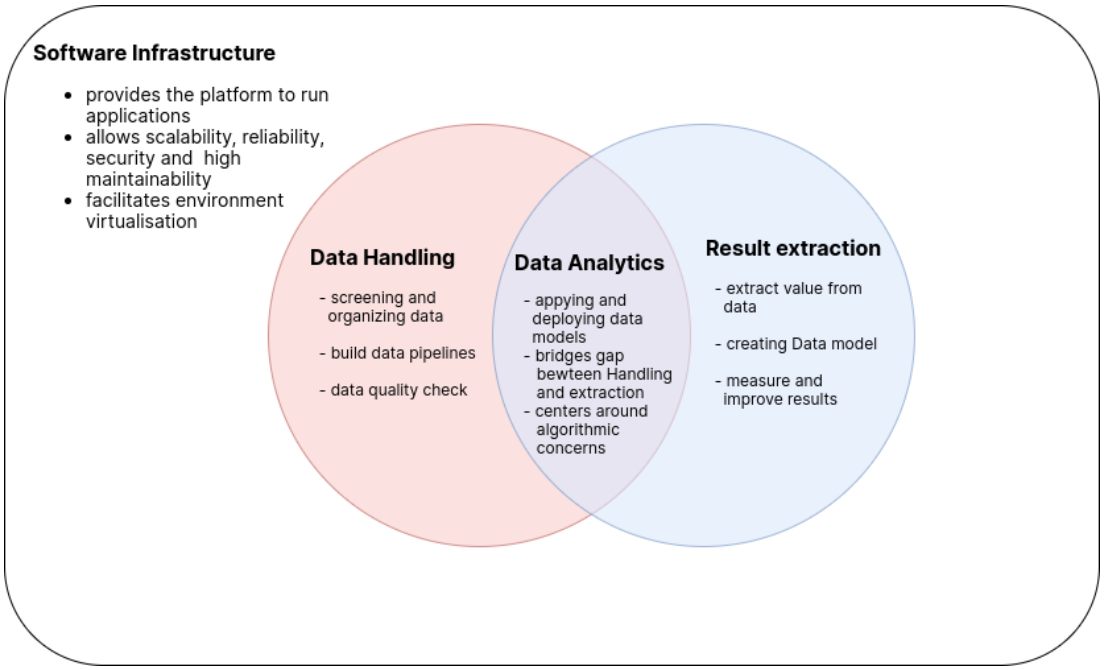


Figure 2.10: We structured the data concern into the fields: Data Handling, Data Analytics and Result extraction. These areas have their distinct focus but can overlap in individual tasks. Above all, these areas depend on a software infrastructure to run all further analyses.

development perspective: unstructured, structured, and semistructured data. First, structured data is predefined modelled data. A data model refers to programmatically defined data that formalities objects and relationships abstractly. This data is usually stored in the relational database. Second, unstructured data refers to data that lacks a data model. Unstructured data can have different formats, e.g. text, audio, image or video files. Therefore, the data is generally stored in its raw form, e.g. a data lake or NoSql databases. Third, semistructured data refers to labelled or tagged data. Tagged data defines specific data characteristics with preset fields. Preset fields separate semantic elements similar to key-value data types. Moreover, preset fields allow a hierarchical structure for the data. Semistructured is generally stored in text files, e.g. CSV, JSON or XML. Generally, data classes concerning structure vary in their concern and storage type.

In biomedical data analysis, organising data, variety is a significant concern. A large part of data is either unstructured or semistructured, e.g. video files or protein

sequences in the form of FASTA file format [136]. More concretely, two challenges come with dealing with semi- and unstructured data: storage and scattered data. First, storing large sets of semi and unstructured data requires an additional layer of organisation. This organisation includes having the possibility to find the data. Much research suggests adding a metadata layer on top of the data. Second, besides organising a large set of data, scattered data is an immense concern in biomedical data analysis. In some instances, biomedical data is located in a vast storage network with various access rights. Moreover, using a wide range of data inputs allows further novel insights into the body of data. Therefore handling the large data variety is an open challenge concerned with making semi- and unstructured data searchable and unifying scattered data to generate value from data.

Data maintenance One of the open challenges concerning data is data maintenance. Associated challenges with open challenges are: Processing large data sets and maintaining data integrity. First, With growing data amount keeping data consistent is an essential part. For instance, a typical inconsistency can be caused by an invalid data type inside a relational database. This inconsistency can break restoring database backups. So manual intervention is needed to fix the issue. Second, besides data consistency, preprocessing is essential for handling extensive data. One essential task is to process large semi- and unstructured data. This process includes transforming data into structured data or creating pipelines/workflows that transform the data. Moreover, using complex computing resources in an analysis workflow presents its challenges. These challenges include facilitating efficient workload scheduling, scaling and handling errors (Suplatov et al., 2016). Moreover, the interplay between software design and workflows/pipelines requires substantial software development expertise. This expertise provides the experience to outweigh the cost and benefits of designing an appropriate solution for biomedical requirements. Biomedical researchers require an independent, consistent software environment for each step in a workflow/pipeline to run their experiments. For this reason, one key goal is to automate and optimise data maintenance rules. So that researchers and clients can access and use data as assets for their analysis.

System failure recovery When running complex systems, system breakdowns are inevitable. Several factors can cause these breakdowns, e.g. uncovered cases

in the source code, client data inconsistencies or even power outages. After such an event, the recreating process can be rather complex since resolving errors and inconsistent states can be challenging. Especially in the case of workflows/pipelines handling audit trails increases the complexity. Therefore, it is essential to have procedures to recover from a disaster in these instances.

Analyics and Result Extraction This section will outline open challenges in the data process concerting analytics and result extraction. First, one essential part of the analysis is how data model. we will outline the key concepts in data modelling and go over the open challenges. Second, concerning result extraction, we will revisit the critical goals of data-driven analysis. Finally, we outline the issues revolving around novel insights.

Datamodling As mentioned above, structuring semi- and unstructured data plays a significant role in the data-driven analysis. One central part is creating a data model from unstructured data. Data modelling includes defining a structure to be processed by software applications. Generally, this process includes taking various data sources and establishing a structure and relationships. In some instances, even performing calculations and validation. Generally, this process is required when integrating relational databases. From a more abstract perspective, this process is part of a strategy pattern in handling data: Extract load and transform (ELT). Extract Load Transform is a data processing process in which data is extracted from one or more sources. Then they are loaded onto the target server (Load) and transformed in it (Transform). Researchers have used ELT for some time to process or integrate raw data. The process proves particularly advantageous when large amounts of data need to be prepared and structured for data analyses. One open challenge is that each ETL process requires a customised approach to process and model data. From a biomedical perspective, each research approach uses a slightly different point of view. These slight variations require adapted data availability. These include a subset of data as well as data models. Therefore, satisfying the highly variable data model to demand is an open challenge since the various approaches require high flexibility.

Novel insights A large amount of data benefits greatly from data-driven anal-

ysis; however, it also requires an additional layer of organisation. As mentioned above, one key goal in data-driven analysis is to discover new results or patterns from existing data. Therefore, two challenges affect the advancement: finding all the needed data and utilising a broad spectrum of tools. It is not easy to handle both challenges since they require a delicate balance of requirements and a customised process.

2.4.3 Barriers to analysing biomedical data

In biomedical data analysis data, sensitivity affects further analysis immensely. When dealing with biomedical data, researchers must consider administrative concerns when conducting further analysis. The two main challenges are sensitive privacy concerns and separated data access and different administrative responsibilities. First, biomedical data includes sensitive patient data subject to data privacy regulation or the agreement between clients and research facilities. These regulations have various restrictions. In some instances, this restriction only allows data analysis on designated servers. Second, in some instances, data is scattered around a network of servers with independent administrative regulations. The open challenge is to create a system that can handle sensitive private data and unify data across the administrative department.

In comparison to software engeneering fields, biomedical software development is still in the early stages while having the same challenges as in the traditional areas of the so-called Big Data applications (e.g. finance or climate) [46]. These include the well-known 4 'V's: volume, velocity, variety and resolution [61]. A good area covering these challenges is sequence analysis with new data-generating methods. For example, Next Generation Sequencing, which aids in decoding human genetics, produces large data sets in genomics. These data sets can range from dozens to hundreds of gigabytes per sample, depending on the application [56, 97].

Nevertheless, in recent years the field of biomedicine has grown significantly. For instance, in genomics, Next Generation Sequencing, which aids in decoding human genetics, produces large data sets. These data sets can range from dozens to hundreds of gigabytes per sample, depending on the application [97]. At the same time, physicians store medical patient records primarily digitally [15, 144], which adds to

the volume and the variety of data. Whereas other fields have integrated new systems to deal with this vast amount of data, biomedical research is still developing [46]. For instance, combining healthcare data with molecular pathology can bring new insights for treatment and create individual healthcare plans [46].

We summarise the challenges: Research facilities and hospitals have large bodies of data that must be stored on-premise with an easily searchable database. Additionally, researchers and scientists require the possibility of using various tools to analyse these extensive data sets.

2.5 Conclusion

In conclusion, the use of distributed computing and containerisation technologies have the potential to revolutionize the field of bioinformatics by enabling the efficient handling of large amounts of data and the development of reliable software systems, scalable, and performant. In this chapter, we have reviewed the literature on software systems and data management systems for bioinformatics data analysis that makes use of these technologies, and identified several open research questions and challenges related to their design and implementation. These include questions about the impact of software patterns on the reliability, scalability, and performance of a bioinformatics software system; the key considerations and challenges in designing and implementing such a system; the role of advanced software engineering principles and practices in supporting the effective integration of data from multiple sources; and the factors that influence the success of such a system and how they can be optimized in the design and implementation process. We have also explored the potential benefits and challenges of applying parallel computation techniques and machine learning techniques to improve the speed and effectiveness of data analysis, and the scalability of our framework for large-scale handling datasets in bioinformatics.

We believe that addressing these research questions and challenges will be critical for advancing the field of bioinformatics and supporting the effective integration and analysis of data from multiple sources. We hope that this review will help to stimulate further research and discussion on these essential topics.

Chapter 3

Advanced Software Engineering in Bioinformatic

A Case Study of Design and Implementation

In this chapter, we will explore the impact of advanced software engineering principles and practices on the efficiency and productivity of a research team working with a bioinformatics software system. We will examine how these practices can improve the team's ability to effectively use and maintain the system, as well as how they can contribute to the overall success of the research project. We will also discuss the key considerations and challenges in designing and implementing a bioinformatics software system that is deployed and maintained on-premise and autonomously while maintaining data security and privacy. Additionally, we will examine how the integration of advanced software engineering principles and practices can support the effective integration of data from multiple sources in a bioinformatics software system. Through this discussion, we hope to provide insights and guidance for those seeking to design and build bioinformatics software systems that are scalable, maintainable, and able to handle large amounts of data in an on-premise, academic setting.

3.1 Introduction

The integration of advanced software engineering practices, such as the use of software patterns, containerisation, and distributed computing, is crucial for the design and implementation of bioinformatics software systems. These systems are used to analyze and interpret biological data, often require high levels of reliability, scalability and performance, and can benefit from the use of techniques such as containerized virtualization using Docker. To our knowledge, there is currently no framework or system that utilizes a state-of-the-art service-oriented architecture (SOA) for medical data analysis.

Biomedical scientists and researchers generally use analysis tools to solve research questions [159]. However, usability, including understandability, learnability, operability, and attractiveness is often a secondary or omitted consideration in the research [153]. This chapter aims to make usability is a central topic, as it has several advantages for developers and users. Improved usability can facilitate faster issue resolution and the implementation of more features allows users to implement data analysis algorithms faster, improve morale in working with the framework, and allow developers to address technical debt efficiently [44, 167].

In addition to usability, it is crucial for medical software to be adaptive to a variety of demands, such as different tools versions and environments. A flexible software architecture with an independent environment, such as a service-oriented architecture or containerized the environment can prevent the rework caused by technical debt. Docker is a containerized, lightweight virtualization technology with a wide range of applications, including developing, packaging, shipping, deploying and executing applications into containers. Docker has several benefits over traditional virtual machines (VMs), including the consumption of fewer resources, equal or better performance, small image size, and the ability to roll back to previous versions.

Elasticity, or the provisioning of resources and computing instances, is also an essential consideration in the design of bioinformatics software systems. Vertical elasticity is the adaptation of resources, such as CPU and memory, assigned to an instance. In contrast, horizontal elasticity refers to creating or destroying instances of computing resources associated with an application. Elasticity can be achieved through automation and optimization, resulting in scalability or sustaining heavy workloads using additional resources. In this chapter, we will investigate the impact of elasticity on the efficiency and productivity of a small research group, as well as the key considerations and challenges in designing and implementing an elastic system.

These challenges can be summarised in the following research question:

- How does the use of software patterns in the design of a bioinformatics software system impact its reliability, scalability, and performance?
- What are the key considerations and challenges in designing and implementing a bioinformatics software system with Docker, microservices, and distributed computing?
- How does the integration of advanced software engineering principles and practices into a bioinformatics software system support the effective integration of data from multiple sources?
- How does the adoption of a distributed architecture for a bioinformatics software systems impact the efficiency and productivity of a small research group?
- What are the key factors that influence the success of a bioinformatics software system with Docker, microservices, and distributed computing, and how these

can be optimised in the design and implementation process?

- What are the challenges and potential solutions for the implementation of container orchestration in bioinformatics research teams with limited resources, and what are the benefits and trade-offs of using container orchestration in such a context?

To address these research questions, we will conduct a case study of a a bioinformatics software system that has been designed and implemented with Docker, microservices, and distributed computing. The system's architecture and software patterns will be analysed, and their impact on the system's reliability, scalability, and performance will be evaluated. The experiences and perspectives of researchers who have used the system will also be considered, and any challenges or limitations encountered during the development and maintenance process will be explored.

In summary, this chapter aims to contribute to our understanding of the role of advanced software engineering principles and practices, including general software patterns, Docker, and distributed computing in the design and implementation of bioinformatics software systems, and to provide practical insights and recommendations for researchers and practitioners are working in this field.

3.2 Background: Technical Considerations for Bioinformatics Software Development

The background section of this chapter will focus on the software principles and technical aspects of bioinformatics software systems — including software architecture principles and the use of microservices, Docker, object storage, and representational state transfer. We will begin by discussing the concept of technical debt and its impact on the design and development of software systems. We will then delve into the principles of software architecture. This examination includes the importance of modularity, maintainability, and scalability. Next, we will introduce the concept of microservice architecture and its benefits for building large-scale, distributed sys-

tems. We will also provide an overview of Docker. Docker is a popular tool for packaging and deploying applications in a containerised environment. Additionally, we will discuss the use of object storage for storing and managing large amounts of data in a distributed system and the role of representational state transferin enabling communication between microservices. By understanding these technical concepts, we aim to provide a foundation for discussing the design and implementation of bioinformatics software systems in the context of reliability, scalability, and performance.

Software Engineering principles

Software engineering principles play a crucial role in the development of robust, scalable, and maintainable software applications. These principles include technical debt management, which involves carefully balancing short-term convenience with long-term sustainability in order to avoid accumulating unnecessary costs over time. Software architecture, the high-level structure of a software system, is also a the fundamental principle, as it determines how the different components of the system interact and work together. Microservice architecture, a a pattern that involves decomposing a monolithic application into a set of more minor, independent services is another essential principle that can improve the modularity, scalability, and maintainability of a the software system, but it also requires careful planning and management [127]. Together, these principles form the foundation for effective software engineering practices and the creation of high-quality software.

3.2.1 Technical debt

Biomedical software engineers play a crucial role in developing and maintaining software for the biomedical domain. However, it has been observed that many biomedical scientists often need to pay more attention to the principles of software engineering in their work [159]. This situation can lead to significant challenges in the ability to enhance and maintain software effectively, such as reduced reliability, scalability, and flexibility of the software [26]. By ignoring software engineering principles, the risk of technical debt and the need for rework at a later stage in the project also increases.

Therefore, it is essential for biomedical software engineers to consider these principles when designing software to ensure the creation of high-quality, maintainable software.

One crucial aspect of software engineering principles is the concept of technical debt. Technical debt refers to the additional cost and effort required to fix problems or improve a system due to adopting a short-term, quick solution rather than a well-thought-out approach [167]. In other words, technical debt occurs when software is developed without considering guiding software engineering principles, which can lead to the need for rework at a later stage in the project. Therefore, it is crucial for biomedical software engineers to consider technical debt in their work to avoid unnecessary additional effort in the future.

Technical debt can be considered similar to financial debt in many ways. For example, when an individual takes on financial debt, they must make regular payments. Therefore, this individual must consider two main challenges: first, accurately estimating the monthly debt payment amount, i.e. determining the ratio of their income that can be used for the monthly payments; and second, the accumulation of late payment interest if the debt payments are consistently neglected. This interest in the debt can ultimately lead to bankruptcy. Similarly, technical debt can lead to a state known as technical bankruptcy, where further development becomes very costly or even impossible. Therefore, it is vital to address technical debt promptly and appropriately to avoid negative consequences such as technical bankruptcy.

As with financial debt, it is crucial to address technical debt to avoid adverse effects regularly. For instance, ignoring technical debt can result in a high cost of change, as described by Suryanarayana et al. [167]. The cost of change refers to the time or effort required to change or add a feature to the software. A high cost of change can have several negative impacts on a software project. These negative impacts may include making it difficult to understand certain parts of the code and increasing the effort and time required to implement features and fixes. Moreover, it can reduce reusability and demoralise the software development team. These adverse effects can slow down the progress of a software project. Bas et al. estimate that the requirement and code analysis phase takes up at least 78% of the total time from start to finish in a software project [38]. Therefore, it is crucial to design easily

understandable software to facilitate the development process. Therefore, to avoid technical bankruptcy, it is essential to consider technical debt from the beginning and continuously reflect on technical decisions throughout the software development process.

Now that we have examined the consequences of technical debt let's turn our attention to the causes of this phenomenon. According to research, there are two primary causes of technical debt: a lack of knowledge and skills in software development fields, and a lack of experience in applying software engineering principles [26, 167].

First, a lack of knowledge in relevant software fields, such as design patterns and clean code principles can lead to technical debt. Second, a lack of experience applying these principles can also result in technical debt, as choosing the appropriate design style for a given situation requires experience. Finally, time pressure is a common factor leading to technical debt. Specifically, the pressure to produce results quickly may lead to the selection of a quick but flawed solution, which can result in technical debt. Therefore, a lack of knowledge and experience is a significant cause of technical debt.

To further understand technical debt, Suryanarayana et al. [167] proposed a classification system that divides technical debt into several subcategories. These include

1. design debt (violations of design rules when building the software),
2. code debt (inconsistent code style or the absence of design patterns),
3. test debt (a lack of proper testing), and
4. documentation debt (a lack of documentation or outdated documentation).

These types of technical debt correspond to different phases of the software development process. For developers to minimise the cost of addressing technical debt, it is essential to focus on the software process's most efficient and beneficial phase. For example, Clutterbuck et al. found that the cost of changing features after implementation is at least six times higher than during the requirements phase [39]. In large-scale systems, this cost can increase up to 125 times. Therefore, this thesis will focus on design debt, which occurs during the early stages of the software development process.

In conclusion, technical debt is a common problem in software development that can have significant negative impacts on a project, such as increased costs and reduced reliability. The causes of technical debt includes a lack of knowledge and experience in software development principles and time pressure leading to the adoption of quick but flawed solutions. By understanding the types and causes of technical debt, software development teams can take proactive measures to avoid or minimise technical debt and create high-quality, maintainable software. By continuously reflecting on technical decisions and prioritising the most efficient and beneficial phases of the software development process, teams can effectively address technical debt and ensure the success of their projects.

3.2.2 Software architecture

Technical debt is essential to building a software system and should be considered when outlining the software. Therefore, to build a reliable and expandable software system, it is essential to keep fundamental software architecture principles in mind. Software architecture plays a crucial role in the design and development of software systems, including those used in bioinformatics. Challenges in software architecture include managing complexity, ensuring scalability and promoting reusability. In bioinformatics, these challenges are often compounded by the volume and complexity of the data being analyzed, as well as the need to integrate data from multiple sources and formats. To address these challenges, software architects in the field of bioinformatics must carefully design systems that can handle large, complex data sets, provide tools for integrating and comparing data from different sources, and keep up with rapid advances in technology and biological research. A well-designed software architecture is essential for building scalable, maintainable, and reusable software systems that can effectively support the complex work of bioinformatics.

In order to effectively support the complex work of bioinformatics, it is essential for software architecture to possess several key features: scalability, maintainability, reusability, integration and flexibility. Firstly, scalability refers to the software's ability to handle large and complex data sets and to scale up as needed to meet increased demand or workload. Secondly, maintainability involves having a clear structure and modular design that facilitates understanding and modification of the

system over time. Thirdly, reusability allows components and functionality to be easily repurposed in different contexts or projects. Fourthly, integration provides tools and mechanisms for integrating and comparing data from multiple sources and formats. Finally, flexibility involves adapting to new technologies and advances in biological research as they emerge. By designing software systems with these architectural features in mind, software architects can create scalable, maintainable, and reusable systems that effectively support the needs of bioinformatics research and applications.

Bass et al. define Software architecture as [38]:

> [...] software architecture of a computing system is the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both.

In other words, software architecture refers to the decomposition into parts or components and their relationships. For example, Before any code is written or technical requirements are determined, ideas and requirements are often drawn out when beginning a software project. This phase involves formalizing these ideas into technical components, a process that Richardson refers to as decomposition [147]. This decomposition has two main benefits: it allows for a clear division of labour and responsibilities among team members, and it helps to facilitate a a better understanding of the software as a whole by providing a clear structure of components and their relationships.

The importance of software architecture is closely tied to the requirements of a software application, which typically include both functional and quality of service requirements. Namiot et al. define microservices as "lightweight and independent services that perform single functions collaborating with other similar services through a well-defined interface" [125].

Functional requirements are a fundamental aspect of software development, as they describe the necessary functions that an application must fulfil to meet its intended users' needs. While it is possible to implement functional requirements without considering software architecture, doing so may result in an application that lacks the necessary qualities of service, or QoS, to effectively meet the needs of its

users. QoS is a set of attributes that determine an application's overall effectiveness and efficiency and includes "ilities" such as reliability, scalability, and security. Using software architectureis essential in addressing QoS requirements, providing the necessary concepts and principles to satisfy these attributes effectively.

To better understand the role of software architecturein meeting functional and QoS requirements, Krutchen's descriptive view model [95] can be helpful. This model divides the aspects of software into different views, each of which serves a specific purpose and caters to the needs of different stakeholders. The four views identified by Krutchen are as shown in figure 3.1

1. The logical view, which is designed for end users and includes functionality examples such as UML diagrams, Use case diagrams, Activity diagrams, Sequence diagrams, and State diagrams:
2. The development view, which is meant for the software manager and includes details about the implementation, e.g. Class diagrams, Sequence diagrams, Deployment diagrams, State diagrams.
3. The process view, which explains relationships and typical workflows within the system, e.g. Activity diagrams, Sequence diagrams: , State diagrams: Workflow diagrams:
4. The physical view, which describes the topology of the system, including hardware details and the deployment of each service, eg. Deployment diagrams, Network diagrams, System context diagrams, Physical data flow diagrams

It is common for there to be overlap in the choice of diagrams that can be used to illustrate different views in Krutchen's descriptive view model. This overlap is because software systems are complex and multifaceted, and different stakeholders may have different needs and goals when it comes to an understanding the system.

For example, both sequence diagrams and state diagrams can be used in both the logical view (which provides an overview of the functionality of the system from the perspective of the end-user) and the development view (which provides details about the implementation of the system from the perspective of the software manager). This overlap is possible because both of these diagram types can be useful in illustrating the behaviour of the system from different perspectives.

Similarly, deployment diagrams can be used in both the physical view (which provides an overview of the topology of the system) and the development view (which provides details about the implementation of the system). This overlap is possible because deployment diagrams can provide information about both the physical deployment of the system. Furthermore, the hardware and software components used in its implementation.

Generally, the overlap in the choice of diagrams for different views in Krutchen's model is a reflection of the complexity and multifaceted nature of software systems. By using a combination of different views and diagrams, stakeholders can get a more comprehensive understanding of the system and its various aspects.

Coming back to the general application of Krutchen's different views, it is clear that these views provide a valuable method for describing essential software components from various perspectives. Considering the analogy of building architecture, where different blueprints (e.g. electrical, water, floor, brickwork) are used for different tasks, it becomes clear how Krutchen's views serve a similar purpose. These views provide clear and concise descriptions of software components from different perspectives, such as the end-user, the software manager, and the physical deployment of the system. This approach allows for a more comprehensive understanding of the software system and its various aspects and can be particularly useful in large or complex systems. Overall, Krutchen's descriptive view model offers a valuable tool for stakeholders to understand and work with software systems in a clear and organized way.

Software architecture plays a critical role in the design and development of software systems, including those used in the field of bioinformatics. It is essential to keep critical principles of software architecture in mind when building a reliable and expandable system, including the need to manage complexity, ensure scalability, and promote reusability. In bioinformatics, these challenges are often compounded by the volume and complexity of the data being analyzed, as well as the need to integrate data from multiple sources and formats. To address these challenges, software architects in the field of bioinformatics must carefully design systems that can handle large, complex data sets, provide tools for integrating and comparing data from different sources, and keep up with rapid advances in technology and biological

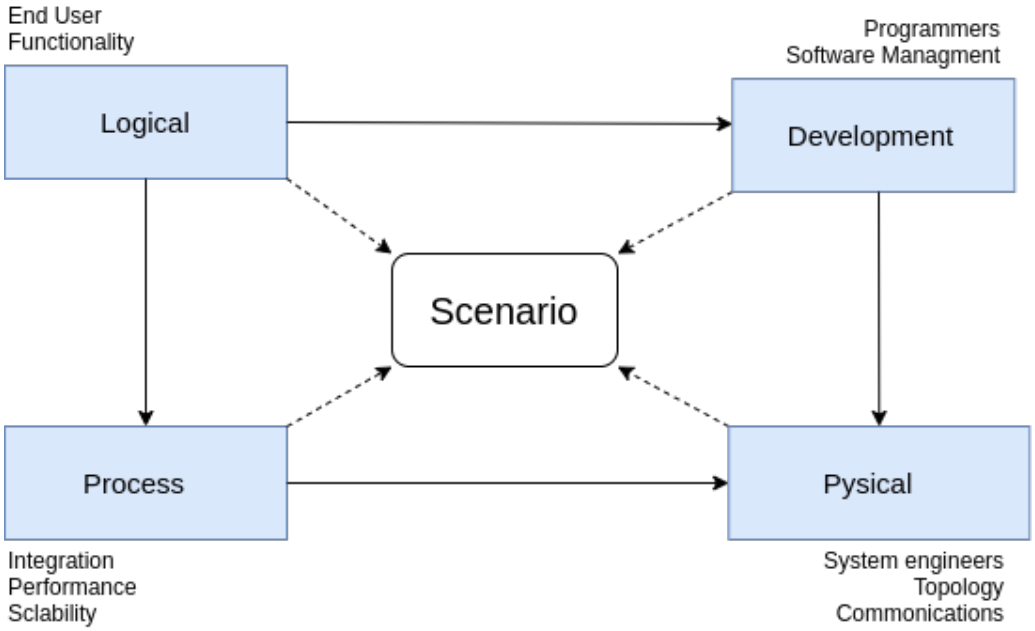


Figure 3.1: Krutchen's 4+1 diagram is a visual representation of the four views in his descriptive view model: the logical view, the development view, the process view, and the physical view, with the scenario view or use cases represented in the center. Depending on the scope and stakeholders, the scenario view may be represented as either a single scenario or a set of use cases. Each of the views is represented as a quadrant surrounding the scenario view. Each view provides a different perspective on the software architecture, with the logical view illustrating the functionality from the perspective of the end user, the development view provides implementation details from the the perspective of the software manager, the process view showing relationships and workflows, and the physical view depicting the topology of the system. The scenario view represents the interactions between the different views and illustrates how they relate to each other. The 4+1 diagram is a useful tool for understanding the different aspects of software architecture and how they relate to each other.

research.

When selecting the appropriate software architecture for a particular the project, it is crucial to consider the specific needs and goals of the system, as well as the expertise and resources available to the development team. One famous architecture

is Microservice architecture. Microservice architecture, which involves dividing the system into small, independent services that communicate with each other through well-defined interfaces is a popular choice due to its ability to scale and maintain individual components independently. However, monolithic architecture, which involves building the system as a single, comprehensive application with all components tightly coupled may also be considered, although it can be more difficult to scale and maintain in the long term compared to microservice architecture. Krutchen's view is that a clear and concise description of the software, architecture is vital for helping stakeholders understand the overall design and functionality of the system, as well as for facilitating communication and collaboration among the development team.

3.2.3 Distributed computing

Several challenges must be considered from a software architecture and bioinformatics standpoint when designing and implementing large-scale data processing and analysis systems. One major challenge is the need to handle and process large amounts of data efficiently, often requiring specialized algorithms and data structures. Another challenge is the need to ensure the accuracy and reliability of the results, mainly when dealing with complex or ambiguous data. Additionally, there may be issues related to data security and privacy, mainly when working with sensitive or confidential data. Finally, there is often a need to integrate various software components and tools to provide a seamless and user-friendly user experience. All of these challenges must be carefully considered to effectively design and implement systems that can handle the demands of large-scale data processing and analysis demands.

In the field of bioinformatics, distributed computing can be used to address the challenges of large-scale data processing and analysis. By distributing data and workloads across multiple computers, distributed computing systems can efficiently process large amounts of biological data, such as genomic sequences, protein structures, and gene expression data. Additionally, the ability of distributed computing systems to be highly available and resilient can be particularly important in bioinformatics, as it ensures that analysis can continue even in the event of component failures or disruptions. The ability to quickly scale distributed computing systems

by adding more computers to the system also makes them well-suited for handling the increasing amounts of data generated by advances in biological research. Overall, distributed computing plays a vital role in the field of bioinformatics, enabling the efficient and effective analysis of large amounts of biological data.

Distributed computing is a field of computer science that involves using multiple computers connected through a network to solve a common problem or perform a shared task. In a distributed computing system, each computer, also known as a node, works together with the other nodes to perform the task at hand. The nodes communicate with each other and share data and workloads to complete the task efficiently. Distributed computing systems can be designed to be highly available, meaning that they can continue to operate even if one or more nodes fail. They can also be easily scaled by adding more nodes to the system, which enables them to handle increased workloads or demand. Finally, distributed computing systems can be designed to be resilient, meaning that they can recover from failures or disruptions without losing data or functionality. Overall, distributed computing allows for the efficient and effective processing of large amounts of data by distributing the workload among multiple nodes.

Several key components are typically involved in distributed computing systems. These include the client, which initiates the request for computation; the server, which response to the request and coordinates the distribution of workloads among the nodes; and the nodes themselves, which perform the computation and return the results to the server. The server is responsible for dividing the workload into smaller tasks, assigning these tasks to the nodes, and collecting and aggregating the results. This process is known as task parallelism, and it enables distributed computing systems to efficiently process large amounts of data by distributing the workload among multiple nodes.

3.2.4 Cloud technologies

Some several challenges and problems can arise when developing and deploying a bioinformatics framework. Some of these challenges include: handling large amounts of data, ensuring the security of sensitive data, meeting compliance requirements, ensuring accessibility, integration with other systems and data sources, and managing

complexity. Each of these challenges can present significant challenges for bioinformatics professionals and finding effective solutions to these problems are critical for the success of any bioinformatics project.

Using integrated cloud technologies in a bioinformatics framework can help to solve many of these challenges. Cloud technologies offer a number of critical features, including scalability, security, accessibility, integration, and disaster recovery, which can be used to improve the functionality and effectiveness of a bioinformatics framework. By leveraging these technologies, it is possible to create a bioinformatics framework that is capable of handling large amounts of data, ensuring the security of sensitive information, meeting compliance requirements, and providing flexible and reliable access to data and computational resources. Additionally, the integration and disaster recovery capabilities of cloud technologies can help to ensure that the bioinformatics framework can operate smoothly and effectively in a range of different environments.

Cloud technologies are a form of computing infrastructure that allows organizations to access and utilize computing resources over the internet. These resources include servers, storage, networking, applications, and other services. In addition, cloud technologies can host various applications and services, including bioinformatics frameworks.

One of the fundamental principles of cloud technologies is elasticity, which refers to the ability to scale up or down to meet changing workloads or data volumes. This feature is handy for bioinformatics applications, which often require significant computational resources and must handle large amounts of data. Other essential principles of cloud technologies include the use of a pay-as-you-go pricing model, the ability to access resources from any location with an internet connection, and the implementation of robust security measures to protect data and ensure compliance with industry regulations. Together, these principles make cloud technologies a flexible and cost-effective solutions for bioinformatics organizations.

3.2.5 Containerisation

In bioinformatics, software engineering challenges such as managing complex dependencies and deploying applications on multiple platforms are common. Bioin-

formatics applications often depend on specific versions of libraries and frameworks for scientific computing, data analysis, or visualization, which can be challenging to manage. Additionally, these applications may need to be run on a variety of different operating systems or platforms, such as Linux, Windows or macOS. This variety can make it challenging to ensure that an application is compatible with all necessary dependencies and can run in all desired environments. Another challenge in bioinformatics is the need to scale and manage applications as they grow and become more complex. As more data is generated and more users access the application, it cannot be easy to ensure that it can handle the increased load and remain stable and reliable. Security is also a concern in bioinformatics, as sensitive data and intellectual property may be at risk if an application is compromised.

Containers can help to address the challenges faced in the field of bioinformatics by providing several useful features. By allowing developers to package an application and all of its dependencies in a single container, containers make it easy to deploy and run an application in any environment, regardless of differences in operating systems or platforms. Containers also provide isolation for the application and its dependencies, which can help prevent conflicts or other issues that can arise when multiple applications are running on the same machine. This isolation can also improve security by limiting the potential impact of a compromise on a single container rather than the entire host machine. In addition, containers make it easy to scale and manage applications, as containers can be easily moved between different hosts or environments without having to worry about compatibility issues. Overall, using containers in bioinformatics can improve the reliability, scalability, and security of software applications in this field.

Containerisation is a technique for packaging and distributing software applications in a way that makes it easy to deploy and run the application in any environment. This infrastructure is achieved by packaging the application and all of its dependencies, including libraries, frameworks, and runtime environments, in a single container. The a container is a self-contained unit that contains everything the an application needs to run, including the necessary libraries, frameworks, and runtime environments. This environment makes it easy to deploy and run the application in any environment, as the container includes everything the application needs to run,

regardless of differences in operating systems or platforms.

One of the most popular tools for containerisation is Docker. Docker is an open-source containerisation platform that makes creating, deploying, and managing containers easy. Docker allows developers to create container images and templates for containers that can be used to create new containers. Docker also provides several tools for managing containers, including container runtime and a container orchestration platform. These tools make it easy to deploy and manage containers at scale, allowing developers to quickly spin up new instances of an application and move containers between different hosts or environments. Overall, Docker is a powerful tool for containerisation that is widely used in the software industry.

3.3 Architecture design

This section presents our bioinformatics data analysis system's critical software architecture design aspects . Our approach is based on microservice architecture, a modular architectural style that allows for developing and deploying individual components as autonomous services. We have chosen to implement this architecture using Docker technology. This choice provides containerisation of our microservices and facilitates their deployment in various environments. To store the data generated by our application, we are using object storage, a highly scalable and durable storage solution. Additionally, we have adopted the Representational State Transfer (REST) the architectural style for component communication, enabling flexible interactions between the various services in our system.

3.3.1 Microservice architecture

Software architecture in bioinformatics is faced with several challenges that must be addressed in order to support them effectively development and maintenance of software solutions in the domain. One of the main challenges is the complexity of the data and algorithms involved in bioinformatics, which often requires advanced computational resources and specialized software tools. Additionally, the rapid pace of innovation in bioinformatics and the constantly evolving nature of the field can

make it difficult to maintain software solutions over time. Moreover, there is a need to support a wide range of clients, including web browsers, mobile devices, and local connections, while also providing a service API with functions such as business logic execution, database access, and request forwarding.

Microservice architecture (microservice architecture) has emerged as a promising approach to addressing these challenges in the context of bioinformatics. By decomposing software solutions into a set of independently deployable and loosely coupled services, microservice architecture enables developers to build and maintain more scalable and flexible software solutions. Additionally, the use of lightweight communication mechanisms, such as HTTP resource APIs, help to facilitate the integration of diverse software components and the reuse of existing resources. Overall, microservice architecture offers several benefits for software development in bioinformatics, including improved scalability, flexibility, and modularity, as well as better support for new and emerging technologies. By separating concerns into independent services, microservice architecture helps to reduce the cost of change and improve the overall quality of the software system while also enabling teams to develop and deploy code independently without affecting other components. Furthermore, the use of loose coupling helps to improve the scalability and availability of the system, as well as other vital qualities such as reliability, testability, and security.

In section 3.2, we discussed the microservice architecture is a style of architecture. In this section, we will highlight microservice architecture key aspects. We outline the context, the benefits and how microservice architecture functions.

In recent years, the use of microservice architecturehas gained increasing popularity as a software design approach in bioinformatics. The complexity and fast-paced evolution of the domain are contributing factors to its popularity. It necessitates software solutions that are adaptable, flexible and modular.

According to Richardson [147], microservice architecture is defined as a style of software architecture that involves the decomposition of an application into independently deployable, loosely coupled services. This approach offers several benefits, including the ability to deploy and scale services independently, the ability to use a diverse set of technologies and programming languages, and the ability to easily update or replace individual services without disrupting the overall system. Addi-

tionally, the use of lightweight communication mechanisms, such as HTTP resource APIs, help to facilitate the integration of diverse software components and the reuse of existing resources.

Fowler and Lewis [67] further refined the definition of microservice architecture by stating that it is an approach to developing a single application as a suite of small services, each running in its own process and communicating with other services through lightweight mechanisms. This approach emphasizes the importance of designing services to be self-contained and focused on a specific business domain, which helps to improve the modularity and maintainability of the system.

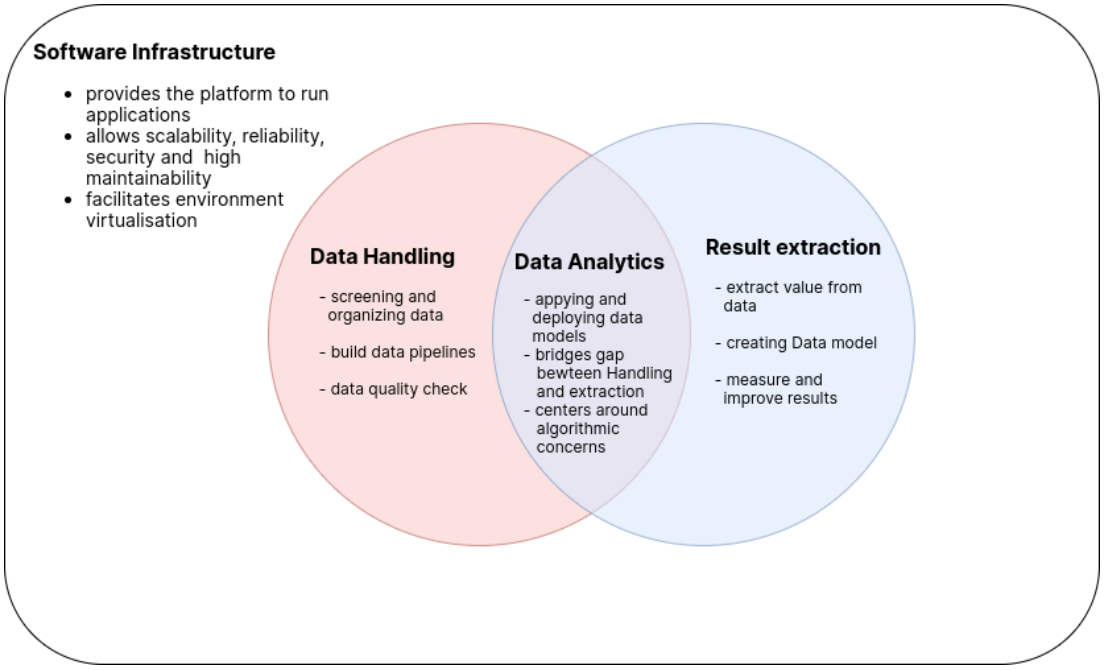


Figure 3.2: Abbott et al. illustrate scalability in the form of a three-dimensional space, with each dimension scaling a different an aspect of an application. The x-axis represents the number of instances, the y-axis represents the functional decomposition and the z-axis represents the data separation. By considering the scalability of an application in the context of this three-dimensional space, it is it possible to understand how microservice architecture (microservice architecture) allows for the independent deployment and scaling of individual services to meet the demand of the system.

The functional decomposition of software solutions characterizes microservice ar-

chitectureinto a set of independently deployable and loosely coupled services. This approach offers several benefits for software development, including improved scalability, flexibility, and modularity. We illustrate the functional decomposition of microservice architecture. It is helpful to compare it to a monolithic application.

In a monolithic application, all services are tightly coupled and must be deployed as a whole. This coupling means that the application is typically scaled along the x-axis (number of instances) and the z-axis (data separation) to meet the demand of the system — refer to figure 3.2. In contrast, microservice architecture includes the functional decomposition (y-axis) to scale the application. This decomposition means that the application is decomposed into a set of independent services, each of which can be deployed and scaled independently to meet the system's demand.

Abbott et al. [14] have further elaborated on this concept by introducing the scalability cube, which provides a three-dimensional space for understanding the functional decomposition of microservice architecture. Along each axis of the scalability cube, a a different aspect of the system is scaled, including the number of instances (x-axis), the functional decomposition (y-axis), and the data separation (z-axis). By considering the functional decomposition of microservice architecture in the context of the scalability cube (as illustrated in Figure 3.2), it is possible to understand how the application is decomposed and how it scales to meet the demands of the system. Overall, the functional decomposition of microservice architecture is a key concept for understanding how it is used to decompose and scale software solutions in bioinformatics. By considering the scalability cube and the functional decomposition along the y-axis, it is possible to understand how microservice architecture allows for the independent deployment and scaling of individual services to meet the demand of the system. This approach offers a number of benefits, including improved scalability, flexibility, and modularity, as well as better support for new and emerging technologies. By separating concerns into independent services, microservice architecture helps to reduce the cost of change and improve the the overall quality of the software system while also enabling teams to develop and deploy code independently without affecting other components. Furthermore, the use of loose coupling helps to improve the scalability and availability of the system, as well as other important qualities such as reliability, testability, and security.

microservice architecture is a software architecture style that functionally decomposes the app into independent services. The decomposition helps in improving the qualities of service, namely scalability, reliability, security, high maintainability, testability, and deployability. This process is supported by pattern language and decomposition strategies.

Pattern language

Several challenges might arise during the development process of building a microservices architecture. One challenge is ensuring that the architecture is designed in a scalable way, meaning it can be easily adapted and expanded as the needs of the system change. This adaption can be challenging to achieve if the architecture is overly complex or inflexible, as it may be difficult to make changes without affecting the system's overall functioning. Another challenge is ensuring that the various microservices can communicate effectively with each other, as this is essential for the smooth functioning of the system. This communication can be challenging if there are issues with the integration of the microservices or if incompatible communication protocols are used. Additionally, building a microservices architecture requires careful consideration of the security and privacy of the system. The decentralized nature of microservices can present additional security risks. Therefore, ensuring that the architecture is secure and that appropriate measures are in place to protect sensitive data can be a significant challenge.

One way to address these challenges is to use pattern language in the development process. Pattern language provides a common vocabulary and structure for communication about design decisions, which can help to ensure that all team members are on the same page and reduce misunderstandings. It also allows for the reuse of proven solutions to recurring problems, saving time and effort in the development process and promoting the sharing of best practices. Additionally, using pattern language can help to ensure that the architecture is structured in a way that allows for scalability, making it easier to adapt and expand the system as needed. Pattern language can be a valuable tool for resolving challenges when developing a microservices architecture.

In a microservices architecture, pattern language is a structured language used

to describe components, and their interactions [69]. It was initially developed for use in building architecture to provide solutions to recurring problems [16] but has since been applied to software development as well [88]. A pattern defines a family of systems in terms of their structural organization and behaviour and consists of several components, including forces, resulting context, and related patterns. The forces component describes the overall problem and any additional or conflicting issues that must be considered when solving it. These issues may conflict and must be weighed following demand and prioritization. The resulting context component describes the consequences of applying a pattern, including any benefits that solve the forces, drawbacks of unresolved forces, and introduced issues. Finally, the related pattern component describes the relationship between the applied pattern and other patterns, such as predecessor, successor, alternative, generalization, and specialization.

In the context of microservices architecture, Richardson has provided a system that includes three categories of patterns: Infrastructure patterns, application infrastructure patterns, and application patterns [69]. Infrastructure patterns address infrastructure issues outside of development and are focused on solving primarily infrastructure-related problems. Application infrastructure patterns, on the other hand, address infrastructure issues that also impact development and are concerned with the intersection of infrastructure and development. Finally, application patterns are used to solve problems faced by developers, such as implementing functionalities or addressing design issues. These patterns provide a way to address the challenges that may arise when developing a microservices architecture, such as ensuring scalability, effective communication between microservices, and appropriate security measures. By using these patterns, software developers can more effectively address the challenges faced when building a microservices architecture, leading to more efficient and effective software development.

Overall, pattern language in software architecture provides a vocabulary and structure for communicating about microservices architecture [69], which can help to facilitate communication and understanding among team members [16], allow for the reuse of proven solutions [88], and ensure that the architecture is scalable and adaptable [69]. By using pattern language, software developers can more effectively address the challenges faced when building a microservices architecture, leading to

more efficient and effective software development.

All in all, the use of a software architecture pattern language in the design and implementation of microservice architectureprovides a standardized vocabulary and organizational structure for effective communication and understanding of the system's components and their interactions.

Decompostion

Software development often faces complexity, scalability, and team autonomy challenges. Complexity can arise in monolithic applications, which tend to become unwieldy and difficult to understand and maintain over time. Scalability can also be challenging, as the tight coupling of components in a monolithic application can make it difficult to scale individual components independently as the demands on the system change. Furthermore, significant development teams may need help with coordination and communication, reducing efficiency and effectiveness. These challenges can hinder the development and maintenance of software systems, leading to suboptimal performance and reduced business value.

Service decomposition is a software design approach that addresses these challenges by creating compact, modular services or components. Breaking down a monolithic application into more focused services makes it easier to understand and maintain the system. Additionally, service decomposition allows for independent scaling of services, making it easier to scale the overall system as needed. Service decomposition also promotes team autonomy, as compact, more focused teams can work on specific services, increasing ownership and responsibility. Overall, service decomposition can lead to more maintainable, scalable, and resilient systems, improving software applications' performance and business value.

Decomposition is a software design approach that involves breaking down a monolithic application into a compact, more modular services or components. There are several strategies for decomposing an application, and these strategies often rely on the experience and expertise of software developers [17]. However, there is no definitive metric or framework that guides the decomposition process, as it heavily depends on the application's specific requirements and circumstances. This challenge is further complicated by the dynamic nature of agile development processes, where

requirements and circumstances may change over time.

One commonly used model for decomposition is the MVC (Model-View-Controller) architecture, which divides the application into three distinct components: data, logic, and presentation. Alternatively, Kecskemeti et al. proposed a decomposition in containers approach [90], which focuses on elasticity and scalability. These strategies and others can be used to decompose an application in order to address specific challenges and improve the overall design and performance of the system.

Finally, there are several ways to decompose the application. Most strategies rely on software developers' experience [30]. No governing metric guides the decomposition process since it heavily relies on the requirements and circumstances. Because of an agile working process, these circumstances might change. Therefore the decomposition has to factor in these changes. Moreover, it takes work to compare projects with each other. One governing model, the MVC, includes data, logic and presentation. Kecskemeti et al. proposed a decomposition in containers [90]. This approach focuses on elasticity and scalability. Therefore, we have chosen this approach as well.

There are several other approaches to decomposing an application, including:

1. Decomposing by business capability and defining services corresponding to business capabilities.
2. Decomposing by domain-driven design subdomain.
3. Decomposing by verb or use case and defining services that are responsible for particular actions, such as a Shipping Service that is responsible for completing shipping orders.
4. Decomposing by noun or resource by defining a service responsible for all operations on entities or resources of a given type, such as an Account Service that is responsible for managing user accounts.

Service decomposition is a software design approach that breaks down a monolithic application into a compact, more modular services or components. This approach is often used in microservices architecture, designing software applications as a set of loosely coupled, independent services that can be developed, deployed, and scaled independently. Service decomposition can help to address challenges such as

complexity, scalability, and team autonomy, leading to more maintainable, scalable, and resilient systems. Various strategies for decomposing an application include decomposition by business capability, domain-driven design subdomain, verb or use case, and noun or resource. In addition, decomposition can be aided by using pattern language and decomposition strategies and is supported by the microservices architecture. However, the decomposition process depends on the application's specific requirements and circumstances, and there is no definitive metric or framework that guides the process.

3.3.2 Docker technology

As mentioned in section *3.2.5 Containerisation*, the field of bioinformatics faces several software engineering challenges, such as managing complex dependencies and deploying applications on multiple platforms. These challenges can be addressed through containers, which provide several useful features. By packaging an application and its dependencies in a single container, developers can quickly deploy and run the application in any environment, regardless of differences in operating systems or platforms. Containers also provide isolation for the application, which can improve security and prevent conflicts when multiple applications are running on the same machine. In addition, containers make it easy to scale and manage applications as they grow in complexity and usage. Overall, using containers in bioinformatics can help to improve the reliability, scalability, and security of software applications in this field.

To implement a microservice architecture for our proposed system, we have chosen to utilize Docker. This technology is a form of containerized virtualization. In order to clarify the concept of container-based virtualization, it is necessary first to understand virtualization itself. Essentially, virtualization refers to the creation of a subsystem, known as a guest, on an existing system, known as a host. These subsystems are often able to provide independent hardware and software environments. Container-based virtualization is a specific type in which independent, software-based subsystems, known as containers, are created. In the context of a microservice architecture, the use of container-based virtualization can help to separate and isolate individual services.

Docker is a containerisation technology widely applied in various contexts, from development to deployment in isolated environments. As a form of lightweight system-level virtualization, it offers several benefits compared to traditional virtual machines (VMs). Containers consume fewer resources than VMs because they share the host system's hypervisor, resulting in greater efficiency and the ability to deploy more containers on a single physical machine. Additionally, containers can separate computing environments from the host system, which helps to avoid conflicts and improves performance. Docker also includes a versioning tool that allows users to roll back to previous versions quickly. Furthermore, containers have a small image size, which makes them faster to generate, distribute, and download, and they require less storage space. These features make Docker a valuable tool for implementing vertical elasticity in autonomic systems, where it can help avoid over-provisioning and under-provisioning problems.

Comparing container-based virtualization to hypervisor-based virtualization, it is possible to identify several benefits of the former approach. While both forms of virtualization involve the separation of different systems from the host system, container-based virtualization differs in that it does not utilize a hypervisor, which is a software layer that provides a virtual operating system for guest systems and manages their execution (see Figure 3.3 for a visual representation of this process). Instead, container-based virtualization relies on a lighter-weight approach, utilizing fewer resources and requiring less disk space due to its lack of reliance on an entire kernel (as discussed in [189]). Later in the text, we will delve further into how containers facilitate resource efficiency. Overall, it can be concluded that the benefits of container-based virtualization include reduced resource consumption compared to hypervisor-based virtualization.

Docker architecture and critical features

Docker is a popular container virtualization technology released as an open-source project in 2013 [146]. It provides a command-line interface for creating, managing, and deploying containers and isolated environments with file systems that can run independently. Docker also has revision control capabilities, allowing containers to be easily shared through repositories such as Docker Hub.

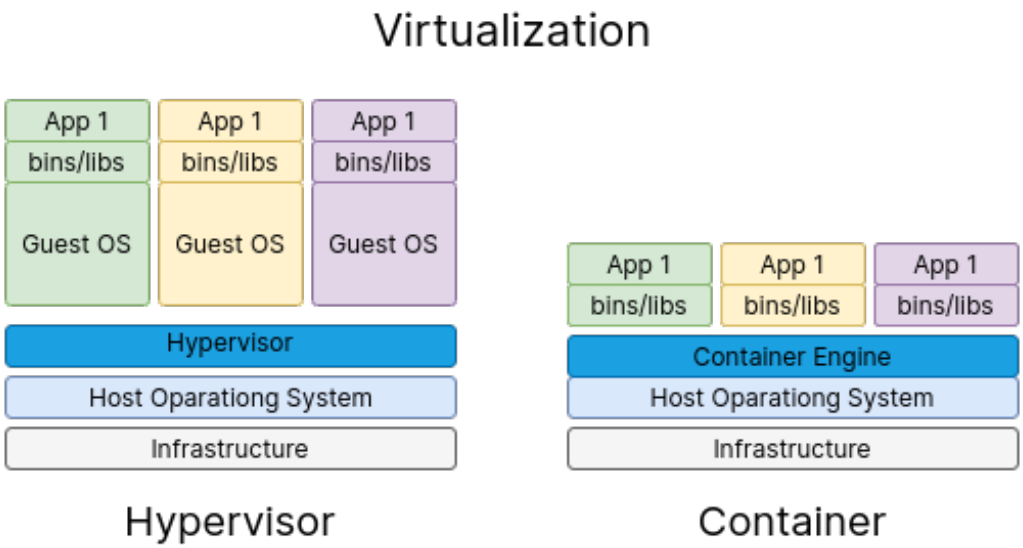


Figure 3.3: In hypervisor-based virtualization, a hypervisor sits between the hardware and the operating system, providing virtual operating systems for multiple guest systems. The hypervisor manages the execution of these guest systems, allowing them to operate independently of the host system. In contrast, container-based virtualization relies on the host operating system and utilizes containers to separate applications and their dependencies into isolated environments. These containers share the host operating system kernel and do not require a hypervisor, resulting in a more lightweight and efficient approach. Overall, this figure illustrates the differences between these two forms of virtualization in terms of their architecture and resource utilization.

The architecture of Docker consists of three main components:

1. Docker Client: This component is the primary way to interact with the Docker system, allowing users to issue commands such as "build" and "run".
2. Docker Host: This component handles core functionalities, including API requests and management of Docker objects like images, containers, networks, and volumes. The Docker daemon, a component of the Docker Host, is responsible for the system's business logic. The Docker Host also stores and manages Docker images and containers.
3. Docker Registry: This component stores Docker images, which can be either publicly available or privately deployed. One example of a widely-used public registry is Docker Hub.

Docker operates using a similar architecture to that of a three-tier application, a type of software architecture organized into three distinct layers. The Docker Client is the presentation layer, responsible for displaying information to the user and receiving input. The Docker Host is the application logic layer, handling core functionalities and processing user requests. The Docker Registry serves as the data storage layer, storing and managing the data used by the application.

In a three-tier architecture, the separation of these functionalities into distinct layers allows for better scalability, maintainability, and security. Similarly, the separation of these responsibilities Docker enables the efficient creation, management, and deployment of containers while also allowing for independent operation and easy sharing through revision control and repositories.

Docker utilizes Linux kernel features such as namespaces and control groups (cgroups) to separate environments and enable efficient resource allocation for containers.

Namespaces partition the operating system's processes into sets or subsets. Each set only has access to a designated set of resources, including IDs, hostnames, and user IDs. These partitions create independent, isolated environments, which Docker uses as the delimitation for its containers. This ensures that each container operates within its sphere of influence and that any modifications made within the container do not affect the host system or other containers. Examples of namespace types that Docker supports include PID, Mount, User, Network, and IPC.

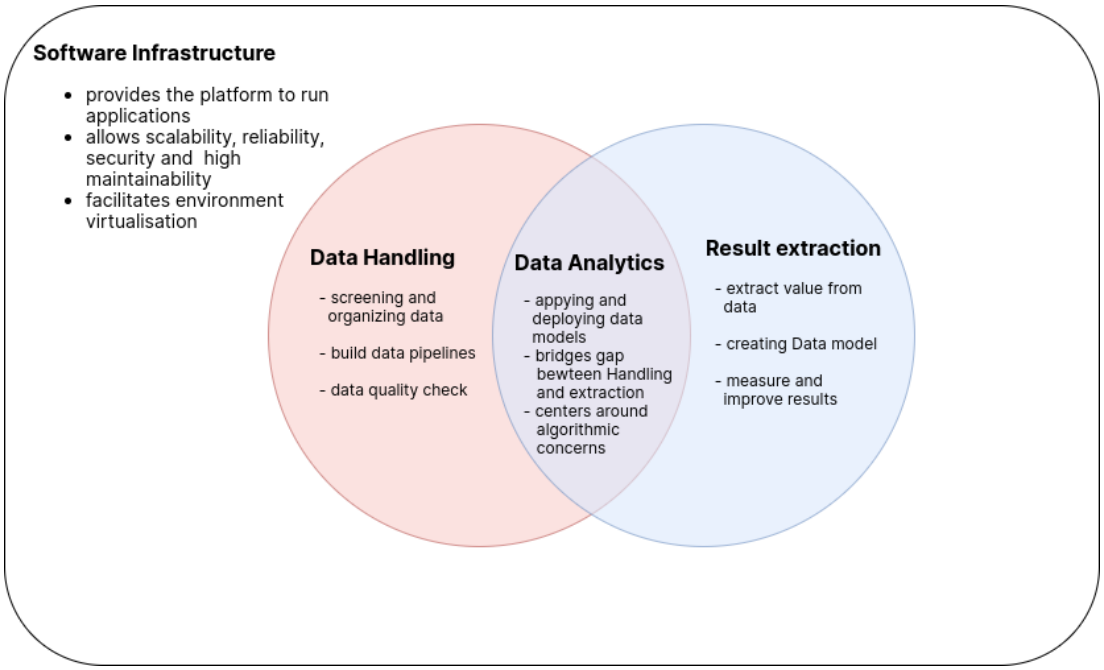


Figure 3.4: Here you see the Docker architecture. Generally, the client sends commands to the docker daemon that executes the commands. With building, an image is created. The pull command receives the image if it is not present. There are several possibilities to handle docker registries. Furthermore, finally, the run command executes the Docker.

Cgroups, on the other hand, manage the resource usage of a group of processes by limiting, tracking, and isolating the resources. They can configure resource limits, prioritize resource usage, and control the status of processes in a group. This makes cgroups an essential aspect of container virtualization, as they allow for the control and management of multiple processes within a container. The use of cgroups allows for more efficient resource allocation in containers than hypervisor virtualization, which requires a complete boot for each virtual machine. As a result, containers, which consume fewer resources than VMs, are often the preferred choice for independent services [51].

In summary, the Linux kernel features of namespaces and cgroups enable Docker to create isolated environments for containers and facilitate efficient resource allocation, making it a popular choice for container virtualization.

This section outlines containerized virtualization and how it helps build microser-

vice architecture. Further, we outlined the Docker architecture and its key features that help in container-based architecture.

3.3.3 Data storage

In the field of bioinformatics, data storage can pose significant challenges. One major challenge is the large volumes of data generated, particularly with the growth of high-throughput sequencing technologies. This challenge can make storing and managing the data difficult and may require the use of distributed storage systems. Another challenge is the complex nature of bioinformatics data, which often has intricate structures and relationships that need to be captured and represented in the data storage solution. In addition, Bioinformatics data is often generated from various sources, requiring integration across different data sets and systems. Ensuring the security and integrity of the data is also essential, as bioinformatics data can be sensitive and may need to be protected from unauthorised access. Finally, bioinformatics data must be easily accessible to researchers and analysts, which can be challenging when dealing with large volumes of data and complex data structures.

Object storage solutions like Minio can help to address the challenges of data storage in the field of bioinformatics from a software architecture perspective [138]. Minio is designed to scale horizontally, storing and managing large volumes of data efficiently. It also stores data as objects, which enables it to handle complex data structures and relationships. In addition, Minio has a RESTful API and supports various programming languages, making it easy to integrate with other applications and systems. It also includes features to secure data, such as encryption at rest and in transit and fine-grained access controls. Minio is also highly performant, allowing for quick and efficient access to data by researchers and analysts. Its support for various protocols and interfaces makes accessing data from various clients easy. Overall, Minio's object storage capabilities make it a valuable tool for addressing data storage challenges in bioinformatics.

Biomedical scientists often have a different focus when analysing their data, so preserving the raw data in its original form is essential. To achieve this, we propose uploading the data directly to an open-source object storage solution Minio [138]. Object storage is a model in which data is organised into three parts: the data

itself, an associated metadata file, and a unique key. The data is stored in a flat namespace, allowing it to be searched and accessed using metadata or the unique key. This benefit contrasts with hierarchical file systems, in which data is organised into directories and files. Object storage solutions, such as Minio, facilitate easy access to biomedical data with extensive metadata.

Several cloud storage providers offer object storage solutions, such as Amazon Simple Storage System (S3), Microsoft Azure, Rackspace Files, and Google Cloud Storage. Among these options, Amazon S3 has a widely used interface, known as the S3 API, which allows developers to interact with the data [118]. To make it easier for developers to use object storage solutions, open-source developers have implemented the S3 API into projects like Minio. This implementation allows Minio offers all the benefits of object storage while also being compatible with the S3 API. To interact with Minio's S3 API, software developers can use various Software Development Kits (SDKs) in programming languages such as Java, Kotlin, Python, and JavaScript.

In our proposed system, we deploy the object storage on-premises, meaning that the user controls and manages the storage. Data is transmitted to the object storage using a generated S3 URL address. Minio provides a temporarily valid unique URL address, allowing the user to upload the data. By deploying the object storage on-premises, we can ensure that sensitive data remains in a trusted environment, as the servers or nodes are secure. Additionally, we can utilise the Amazon S3 commands and API but store and transfer files within a local network rather than relying on a remote data storage solution. This approach can enable greater control over data security, as the data remains within the user's direct control.

3.3.4 Component communication with representational state transfer architecture

Several challenges and problems must be considered when building a bioinformatics framework from a software architecture viewpoint. One challenge is supporting data interoperability, as different software systems may use different data formats and protocols. A standard way for these systems to communicate and exchange data is needed. Another challenge is scalability, as a bioinformatics framework may be

required to process and analyse large volumes of data in real time. The ability to easily add new services or modify existing ones without affecting the overall system is vital. Ease of use is also an important consideration, as the framework should be easy for developers to access and use and for bioinformatics researchers to analyse data. Modularity is another crucial aspect, as the framework should be designed as a set of independent components that can be reused in multiple applications or replaced with alternative implementations. Finally, the framework should be flexible and be used in a wide range of environments and platforms. The ability to access the framework from different client applications can help achieve this flexibility.

A representational state transfer application programming interface(REST API) can help address several of the challenges mentioned when building a bioinformatics framework. A REST API is a type of software architecture that provides a set of guidelines for building web services. It is designed to make it easy to develop and consume web services by providing a simple and flexible interface that is based on the principles of representational state transfer (REST). Some of the key features of a REST API that make it useful for building bioinformatics frameworks and other distributed systems include the use of standard HTTP methods to perform operations on resources, the use of HTTP status codes to indicate the success or failure of a request, the use of HTTP headers to send additional information about a request or response, the use of universal resource identifiers (URIs) to identify resources and distinguish between different types of operations, and the use of resource representations to represent resources and the data associated with them. These features allow a REST API to support a wide range of functionality, data formats, and client applications, making it a powerful tool for building bioinformatics frameworks and other distributed systems.

Technically, a REST API allows different software systems to communicate and exchange data using HTTP, the standard protocol for the web. One of the main features of a REST API is its use of HTTP methods, such as `GET`, POST, PUT, and DELETE, to perform operations on resources. This method allows the API to support a wide range of functionality, such as creating, reading, updating, and deleting resources. The use of HTTP status codes allows the API to provide additional information about the success or failure of a request, making it easier to use and debug.

The use of HTTP headers allows the API to send additional information about a request or response, such as the transmitted data format. The use of universal resource identifiers (URIs) allows the API to identify and distinguish between different types of resources and operations. The use of resource representations allows the API to represent resources and the data associated with them using different formats, such as XML or JSON.

Overall, the features of a REST API make it a robust protocol for building bioinformatics frameworks and other distributed systems, as it provides a simple and flexible way to access and manipulate resources over the web. Since the system components are enclosed in separate environments, they require some form of communication, and using REST makes each component independent of the programming language and platform [31, 92, 115, 145]. REST has become a well-established technology for microservice architecture [115, 145], and its use is widespread in this context.

In addition, our development process employed the use of the Proxy design pattern in conjunction with a load balancer and Data-Driven Routing design pattern. The proxy pattern facilitates the control and management of access to resources, while the load balancer distributes network traffic among servers for improved scalability (figure A.1). The Data-Driven Routing pattern utilizes data to dynamically determine routes, allowing for greater flexibility and adaptability (figure A.2). This combination of patterns improves the robustness, reliability and scalability of the system.

3.4 System Implementation

Much biomedical software is based developed to solve a focused research question. The software, in most cases, is just a tool for achieving that goal so that software engineering principles are sometimes neglected. [159] We focus on crucial principles For software development to make it reliable, futureproof and expandable. One critical aspect that is essential for software design is technical debt. Technical debt is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better

approach that would take longer. for this the reason we decided to use a microservice architecture as opposed to a monolithic architecture.

In the past, monolith architecture was a dominant architectural pattern in designing software. Because all components and services are inside a single application in a monolith architecture, it reduces the system's complexity. However, a monolithic architecture has several drawbacks. For example, scaling is only possible when duplication of the whole system, most services are tightly coupled, and errors and bugs can break the application. In contrast, microservice architecture solves these concerns with these design principles.

1. Independent deployability
2. Horizontal scalability
3. Isolation of failures
4. Decentralisation

Problems/issues

Here we will answer questions such as

- How are the patterns applied?
- How did we decompose the services?
- How is it deployed?
- How is the Quality of service integrated?

We address these issues in this section. Further, we will illustrate the design as well as outline the pattern behind our decisions.

Three main advantages are applying Microservice Architecture design principles: 1. Agility, 2. Scalability, and 3. Resilience. First, agility refers to building, deploying and testing so that each can be written in the best-suited programming language for a given task. Moreover, new versions of existing technologies can be integrated without waiting for other application parts to accept new features. Second, since one develops each microservice independently, resources can be scaled rapidly and efficiently. Theoretically, one can relocate microservices to the best-fitting infrastructure for a

task. Third, resilience refers to independence from other components when failures occur. With a decoupled architecture, which uses asynchronous requests, each part avoids interference with the other.

Microservice Architecture is a specialisation of Distributed Computing. Generally, Distributed Systems consist of the following components — called processors — scattered around in a computer network. These components communicate and orchestrate their task by message passing [96]. Distributed Systems can be built to execute processes simultaneously, similar to Parallel Systems. Parallel Systems share many similarities with Distributed Systems [70]. In contrast, each component has its own (distributed) memory for Distributed Systems, whereas Parallel Systems have access to a shared memory [70].

Approach

Our approach includes: services decomposition by domain, self-contained services with responsibility segregation and using container orchestration to deploy and scale the system

3.4.1 Service Decomposition by Domain

We decided to utilise a microservices architecture for bioinformatics data analysis in developing our framework. However, this approach presents several challenges when decomposing the software system. One major challenge is the need to minimise the cost of change to avoid extensive application restructuring. Additionally, the code's functionality may be dispersed throughout the software, making it challenging to locate and modify specific parts. Tightly coupled components can also make it resource-intensive to integrate or replace new components. Furthermore, tightly coupled components may lead to a focus on knowledge, resulting in a high dependency on developers to retain a comprehensive understanding of the complete system. These challenges must be considered when decomposing services for a microservices architecture in developing our framework.

To address these challenges, we propose using the Domain-Driven Design approach to decompose the application into different services. This approach allows us

to define our requirements and utilise them clearly Model-driven design pattern to guide the decomposition process.

In order to achieve the desired properties of the resulting microservices, we must ensure that each service exhibits the following characteristics:

- Stability, meaning that it is resistant to large effects from changes and minimises the ripple effect on related components [185, 186].
- Cohesion, achieved by grouping together functionality and components that are dependent on each other, as described by the Common Closure Principle [117].
- Loose coupling between components, characterised by their independence from one another and minimal mutual knowledge [135].
- Independence, allowing for independent development and deployment.

By carefully considering these factors during the decomposition process, we can effectively design and implement a microservices architecture that meets our requirements and overcomes the challenges outlined above.

Further, we have adopted the Domain-Driven Design (DDD) pattern in the decomposition of our services. DDD is a conceptual framework that aims to achieve stability, cohesive services, loose coupling, and independent services. One fundamental principle of DDD is Model-driven engineering, which involves dividing the software into distinct spheres of concern. This approach has several benefits: first, the architecture becomes stable due to the clear separation of domains and their respective spheres of influence; second, services become cohesive and loosely coupled; third, a loosely coupled architecture provides resilience, maintainability, and extensibility; and fourth, the system becomes more scalable and predictable. Thus, by applying the principle of Model-driven engineering within the DDD pattern, we can effectively decompose our system in a way that meets all of the requirements above.

We decompose our system into services with the domain-driven design(DDD) pattern to achieve these requirements. domain-driven design provides a conceptual framework to ensure stability, cohesive services, loose coupling and independent services. It addresses the above-mentioned requirements with a set of design patterns and principles. One core principle is Model-driven engineering. For this principle,

the software is separated into spheres of concern. The benefits of this approach are: First, the architecture is stable since the subdomains have a separate sphere of influence. Second, services are cohesive and loosely coupled. Third, Loosely coupled architecture provides resilience, maintainability, and extensibility. Third, Systems become more scalable and predictable. Therefore, Domain-Driven Design meets all our requirements for system decomposition. So, we use the Domain-Driven Design principle of Model-driven engineering to decompose our system to achieve the requirements above.

Model-driven engineering (MDE) is a software development approach that relies on modelling techniques to create software systems [63, 171]. One of the primary methods used in MDE is independent process modelling, an abstract architecture model not tied to any specific technology or implementation. In contrast, process-dependent modelling considers the underlying architecture of the system being developed.

Independent process modelling is closely related to Krutchen's View Model [63], in which the logical view represents the process of independent modelling. When designing our system, we chose to use a microservice architecture, which overlaps with both the logical and development views in Kruchten's model [171]. This high level of abstraction allows the development and logical views to share concerns, which we separated into the following components (illustrated in figure 3.5):

- **Managing Logic**: This component is responsible for the control and management aspects of the system. It also includes a database that stores metadata, which is data that describes other data. This component ensures the system's smooth operation by managing and coordinating its various parts.
- **Data storage**: We use raw data storage to store and manage large amounts of data efficiently. This component is crucial for handling the potentially massive amounts of data that the system may need to process.
- **Worker Cluster**: This subdomain contains the analysis tools central to the system's core domain. These tools are used to analyse and interpret the data being processed.
- **Graphical user interface**: This component provides an interface for users to interact with the system and visualise data. It allows users to access the

various features and functions of the system and provides a way for them to view and understand the data being processed.

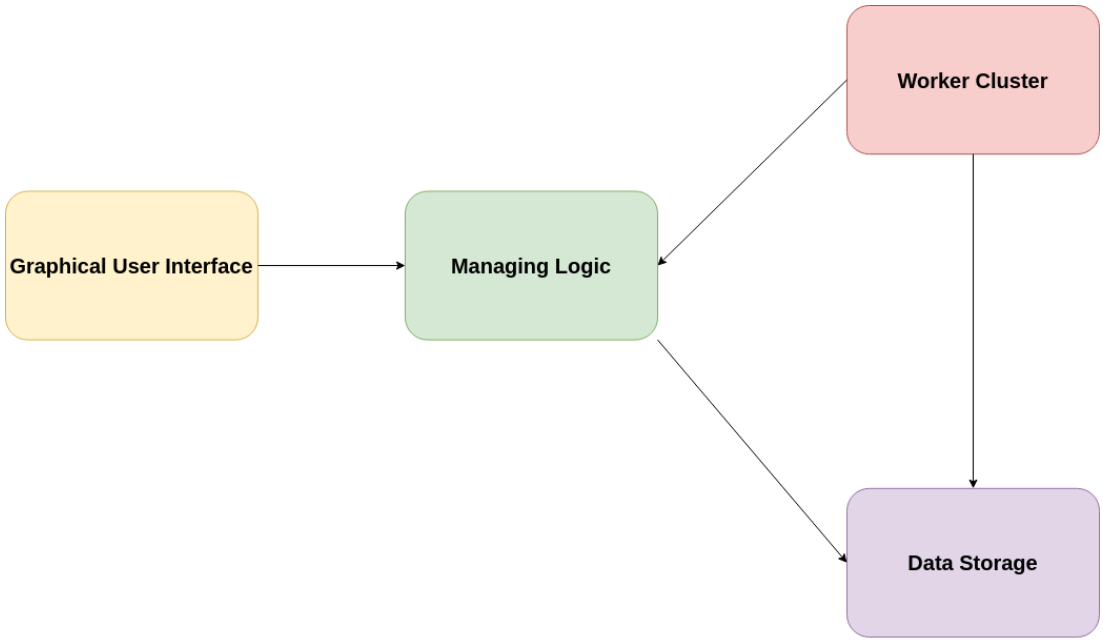


Figure 3.5: We divided the system concerns into four distinct domains. Each domain specialises in one field. The graphical user interface handles the visualisation of data and tools. Further, it provides easy visual control for the functions. The managing logic handles the request and distributes tasks to the worker. The worker cluster handles all the tasks. Finally, the data storage handles all the raw data. With Model-Driven design, this illustration represents Krutchen's logical and development view.

To create a stable, cohesive, and loosely coupled system, we employed a Domain-Driven design to decompose our application. Additionally, we utilised independent process modelling to break down the application into domains. Furthermore, we used a combination of Krutchen's views to visualise the system, allowing for a clear understanding of its structure and behaviour. Overall, these techniques allowed us to design and implement our system effectively.

3.4.2 Self-contained services with responsibility segregation

Model-Driven Design is a popular approach for designing and developing software systems. However, several challenges must be addressed to achieve a successful implementation when applied to a microservice architecture. One such challenge is handling requests and data across a distributed network of microservices. This distribution can be particularly problematic when dealing with large volumes of data or complex data structures that must be passed between services. Another issue is the need to consider the trade-offs between the benefits and drawbacks of using asynchronous data services. While asynchronous communication can improve the scalability and reliability of a system, it can also increase the complexity of the design and make it more difficult to debug issues that arise. Additionally, it is vital to ensure service availability is always maintained, as a single service failure can significantly impact the overall system.

Developers address these challenges by developing various patterns to help improve the loose coupling between independent services and enhance the reliability of the system. One such pattern is the Saga pattern, which allows for the coordination of complex, long-running transactions across a distributed system. Another pattern is the command Query Responsibility Segregation (CQRS), which separates the command and query responsibilities of service to improve scalability and maintainability. The Saga pattern enables an application to maintain data consistency across multiple services without using distributed transactions, while CQRS supports multiple denormalised service views that are scalable and performant and improve the separation of concerns through the use of straightforward command and query models. By utilising these patterns and adequately addressing the challenges of using Model-Driven Design with a microservice architecture is possible to build highly scalable and reliable software systems.

This section will elaborate on the difficulties and drawbacks of using Model-Driven Design with microservice architecture. Further, we will outline the solutions to overcome them. Finally, we will use the two patterns to help solve the loose coupled independent services issue: the Saga and Command Query Responsibility

Segregation.

When introducing a distributed system based on microservice architecture one of the main issues is handling requests and data over a set of services. Further, another problem is to outweigh the benefits and drawbacks of asynchronous data services. At the same time, service availability must be considered when building a system.

Two design patterns help solve this problem. The two patterns that can help solve this problem are Saga and Command Query Responsibility Segregation. First, the Saga pattern enables an application to maintain data consistency across multiple services. The Saga pattern enables data consistency without using distributed transactions. Second, Command Query Responsibility Segregation supports multiple denormalised service views that are scalable and performant. At the same time, Command Query Responsibility Segregation improves the separation of concerns to more straightforward command and query models. The Saga and Command Query Responsibility Segregation pattern facilitate the framework to overcome the added complexity with distributed services.

Saga pattern

The Saga pattern is a well-established design pattern for coordinating complex, long-running transactions in a distributed system [178]. At its core, the Saga pattern involves the use of a central service is referred to as the orchestration-based Saga, which is responsible for managing the events that take place within the system. This central service communicates with the various participants in the system using a command/async reply-style interaction, in which it sends command messages to instruct participants on the actions they should take and receives a reply messages in response. Once a reply message is received, the Saga orchestrator processes the message and determines the next steps in the Saga.

In Figure 3.6, we illustrate how the Saga pattern can be applied in a distributed system. Here, the Managing Logic represents the Saga orchestrator, and the process view, as described by Krutchen, is used to depict the various steps involved in the Saga. As can be seen in the figure, the Saga pattern provides a structured approach for coordinating the activities of multiple participants in a distributed system, enabling the implementation of complex, long-running transactions reliably and

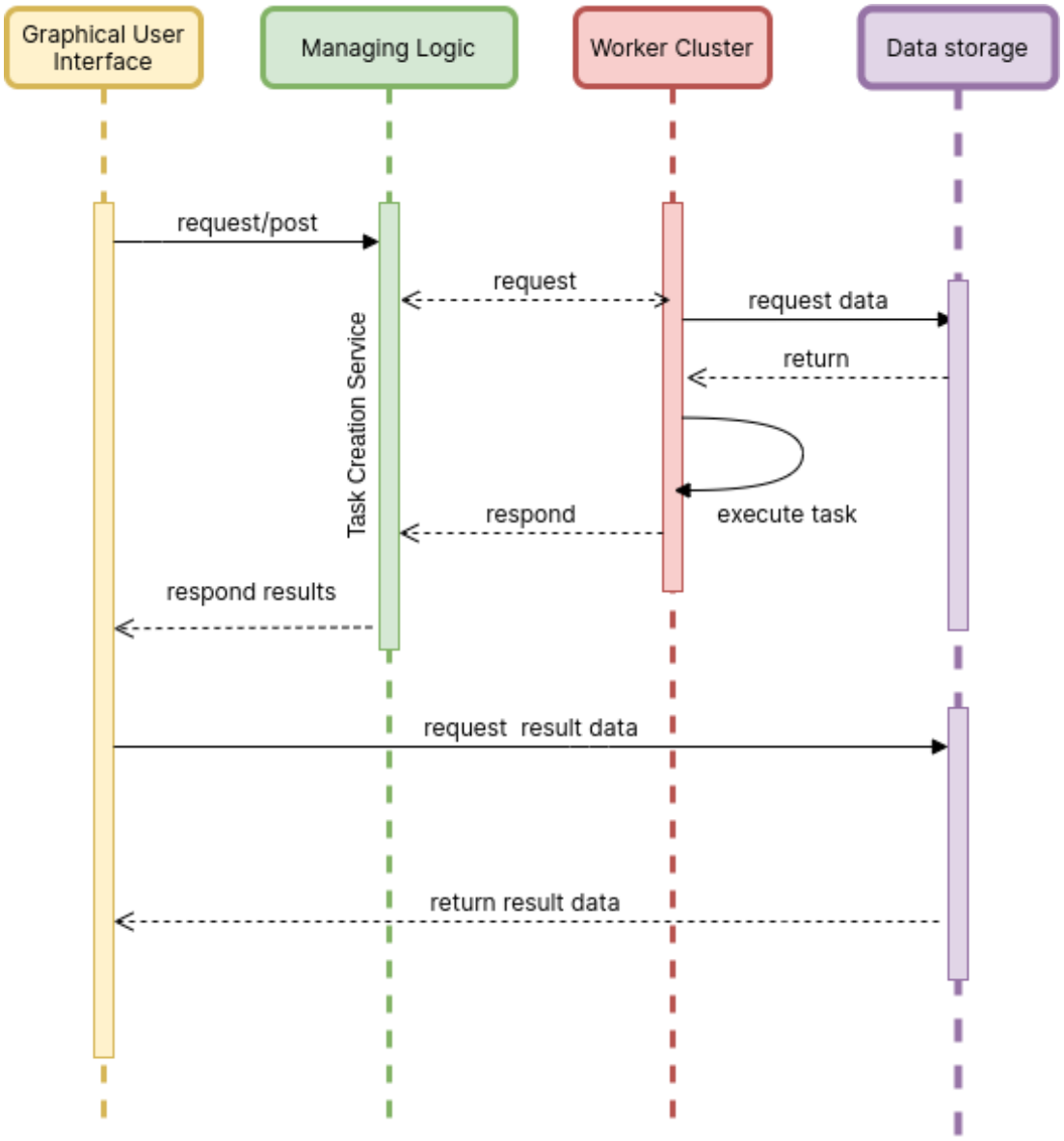


Figure 3.6: The saga orchestration pattern can be visualised using Krutchen's Process view. The Managing logic, which represents the orchestrator, contains all the necessary metadata and executing logic to control the data and assign tasks within the system. The work cluster is provided with executing commands and the relevant data addresses in the data storage, enabling it to carry out its designated tasks. This process view clearly illustrates the orchestrator's role in managing the flow of tasks and data within the system.

efficiently.

Command Query Responsibility Segregation

To further improve the management of data in our system, we implemented the Command Query Responsibility Segregation (CQRS) pattern. As the name suggests, CQRS involves the segregation of concerns through the separation of the persistent data model and related modules into two distinct parts: the command side and the query side. As illustrated in Figure 3.7, the command side is responsible for implementing create, read, update, and delete operations (CRUD), such as HTTP POSTs, GETs, PUTs, and DELETEs, while the query side is responsible for implementing queries and maintaining synchronisation with the command-side data model by subscribing to the events published by the command side. By dividing the data management into these distinct concerns, CQRS enables the system to more effectively divide tasks and improve the scalability and maintainability of the system.

In this section, we presented how the Sage orchestration and command Query Responsibility Segregation pattern can be used to address the drawbacks of implementing a domain-decomposed microservice architecture. By utilising these techniques, it is possible to overcome the challenges that may arise when using this type of architecture, such as ensuring the stability and cohesion of the system and promoting loose coupling between its various components.

3.4.3 Container orchestration

The adoption of self-contained services brings with it several challenges related to management and scalability. Specifically, maintaining and controlling all services at an abstract level can be difficult. Additionally, the ability to scale the services to handle varying workloads, mainly when working with large data sets, is essential. Finally, the organisation of the service deployment is essential to prevent redundant deployment steps for similar services. Therefore, even though self-contained service is a beneficial pattern for structuring a software system. There are further challenges that need to be addressed.

To address management and scalability challenges in using self-contained ser-

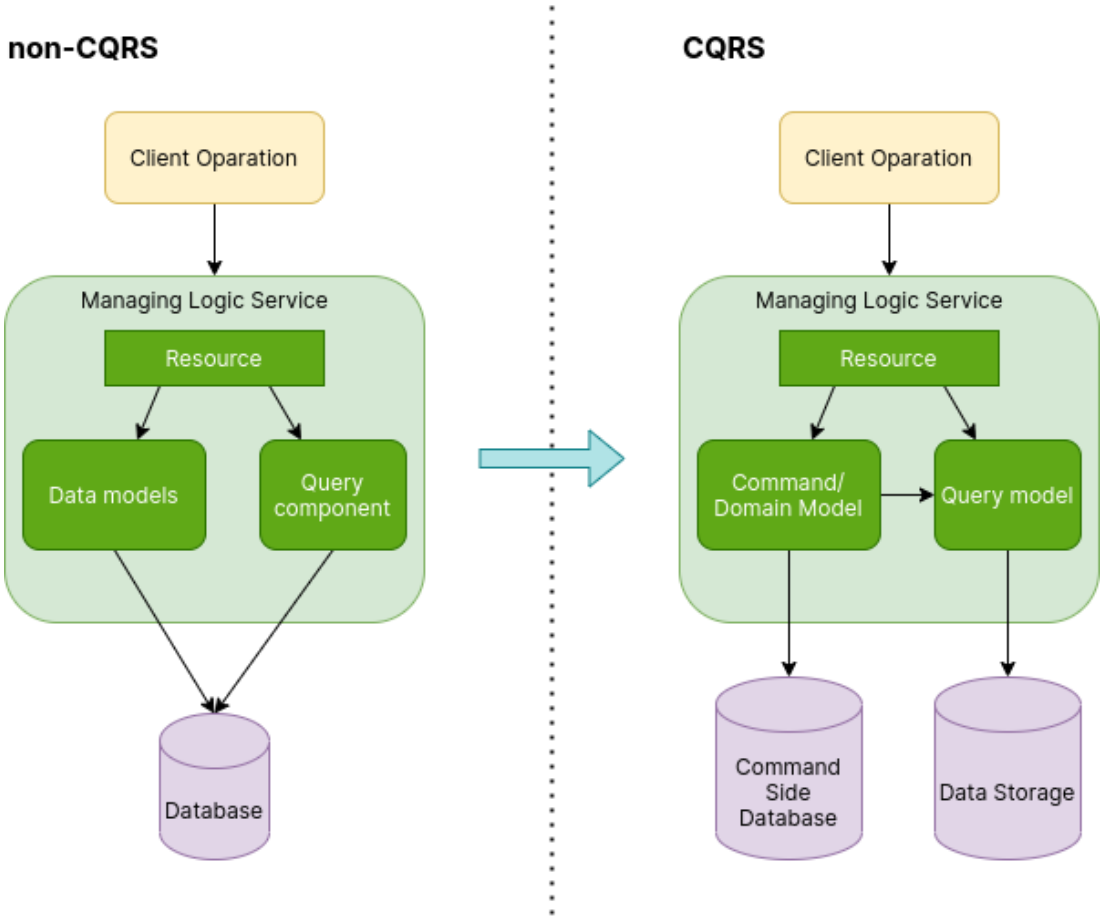


Figure 3.7: The left side of the figure illustrates a non-CQRS architecture in which a single database handles both the command and query functions. The right side depicts a CQRS architecture, which separates the command and query functions into distinct databases. In the CQRS architecture, the command database is responsible for storing and updating data, while the query database retrieves data. This separation of concerns allows for more efficient and flexible handling of data within the system.

vices, we utilise Docker Swarm's integrated orchestration feature. Docker Swarm is a container orchestration platform that manages multiple containers across different host machines, providing an additional layer of abstraction for the management and orchestration of all services. This layer serves as the technical implementation for availability, scalability, and maintainability, allowing for the dynamic reconfiguration of worker nodes and monitoring their status without requiring a complete system rebuild. Additionally, this layer handles all networking routes. In conclusion, using Docker Swarm is a critical component of the technical implementation that enables the realisation of the desired scalability, reliability, security, high maintainability, testability, and deployability [128].

Docker Swarm utilises an abstraction layer to manage and coordinate a set of self-contained services. It allows for the grouping of multiple Docker Engines into a cluster, with each instance, referred to as a node. A node in a swarm cluster is conceptually similar to a standalone Docker node. It can be deployed on a single physical or cloud server or across multiple machines in a production deployment. Each node is assigned one of two roles: manager or worker. The manager is responsible for all administrative tasks, including maintaining the overall state of the cluster, scheduling services, and serving swarm mode endpoints. To ensure consistency and reach a consensus on the state of the cluster, the manager utilises a distributed state machine based on the Raft algorithm [130]. In a Raft assembly with an odd number of managers n, the tolerance for manager failure is $\frac{n-1}{2}$. In contrast, worker nodes are solely responsible for executing containers as instructed by the manager nodes. It is important to note that every swarm cluster must have at least one manager node, which can also be responsible for executing Docker containers in addition to its administrative duties. The separation of roles between manager and worker nodes is depicted in Figure 3.8. Overall, the use of Docker Swarm enables the efficient and effective management of a set of self-contained services, facilitating availability, scalability, and maintainability.

After roles have been assigned to the nodes within a Docker Swarm cluster, application images can be deployed on the cluster using the concept of services. The graphical user interface, general logic, and other components are deployed in our application as different services. The properties of each service, such as replicas and

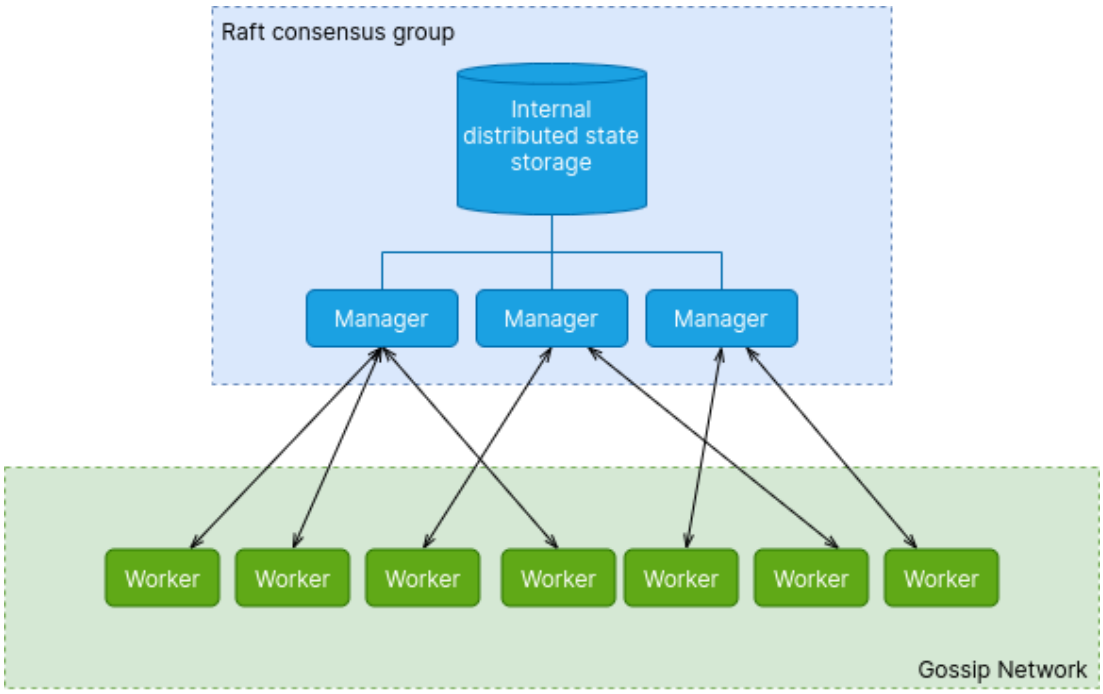


Figure 3.8: This figure illustrates the separation of roles between manager and worker nodes in a Docker Swarm cluster. The top layer depicts the Raft consensus group, which includes an internal distributed state storage and several manager nodes. The manager nodes are responsible for administrative tasks, including maintaining the overall state of the cluster and scheduling services. The bottom layer shows the gossip network, which consists of worker nodes solely responsible for executing containers. The gossip network allows for communication and information sharing among all nodes in the cluster.

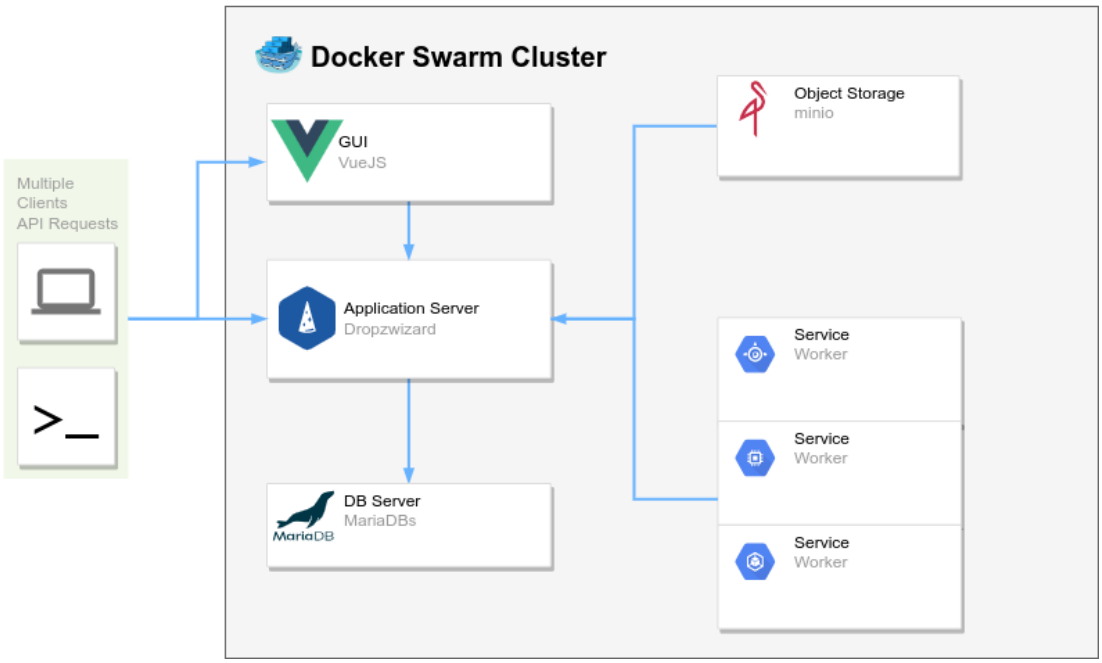


Figure 3.9: This figure illustrates the deployment of the various components of the application within a Docker Swarm cluster. The Physical view shows the topological relationship between these components and how they are integrated within the application. This view can help understand the system's overall architecture and how it is deployed within the cluster.

resource constraints, are defined at the time of deployment. To better understand the deployment of these services within Docker Swarm, we can utilise Krutchen's Physical View — figure 3.9. This perspective presents a topological view of the deployment from a software engineering standpoint, showing the relationships between the various components of the application and how they are deployed on the cluster. This view can help understand the system's overall architecture and how it is deployed within the cluster.

In this section, we outlined how Docker feature swarms solve the issues surrounding scalability and high availability. We outlined how a general cluster works and its essential components.

3.5 Software architecture comparison

The design and implementation of software systems in the field of bioinformatics presents several unique challenges, particularly in terms of ensuring reliability, scalability, and performance. In this study, we sought to investigate the impact of software patterns, Docker, microservices, and distributed computing on the quality and the functionality of bioinformatics software systems. We also examined the key considerations and challenges involved in implementing these technologies in bioinformatics software, as well as the role of advanced software engineering principles and practices in supporting the integration of data from multiple sources. Additionally, we explored the potential benefits and drawbacks of adopting a distributed architecture for bioinformatics software systems in the context of a small research group. In the following discussion, we will present and interpret the results of our research concerning this research questions, and discuss their implications for the design and implementation of bioinformatics software systems.

3.5.1 Summary of results

The objective of this study was to investigate the influence of software patterns, Docker, microservices, and advanced software engineering principles and practices on reliability, scalability, performance, efficiency, and productivity of bioinformatics software systems. Our research findings suggest that the use of software patterns, microservices, and Docker can significantly enhance the reliability, scalability, and performance of these systems.

The modularization of a monolithic system into smaller, independently deployable units through the use of microservices increase the maintainability and modularity of the system, contributing to improved reliability. The adoption of software patterns such as the saga pattern can support the effective integration of data from multiple sources, while the application of the Command Query Responsibility Segregation (CQRS) pattern can enhance the scalability and reliability of the system through the separation of read and write operations. The use of Docker containers for the packaging and deployment of microservices facilitate the scaling and deployment of

the system, leading to improved scalability. In addition, the adoption of object storage for data storage and REST for component communication can further, enhance the performance and reliability of the system.

In terms of advanced software engineering principles and practices, our the research found that the use of model-driven design and Docker orchestration with Swarm can facilitate efficient and productive deployment and management of a large number of Docker containers.

The adoption of a distributed architecture for a bioinformatics software system can further improve the efficiency and productivity of a small research group through the parallelization of tasks and the utilization of additional resources.

In conclusion, our findings suggest that the implementation of software patterns, microservices, Docker, and advanced software engineering principles and practices can significantly enhance the reliability, scalability, performance, efficiency, and productivity of bioinformatics software systems. To optimize the success of a bioinformatics software system with these technologies is important to carefully consider the specific needs and constraints of the system, as well as the availability of resources and expertise.

3.5.2 Interpretation results

The findings of this study suggest that the implementation of software patterns, microservices, Docker, and advanced software engineering principles and practices can significantly improve the reliability, scalability, performance, efficiency, and productivity of bioinformatics software systems.

In terms of reliability, the use of microservices and software patterns such as the saga pattern can enhance the modularity and maintainability of the system, leading to reduced downtime and increased stability. The application of Docker containers for the packaging and deployment of microservices facilitates the scaling and deployment of the system, contributing to improved scalability. The adoption of object storage for data storage and REST for component communication can further enhance the performance and reliability of the system.

The integration of advanced software engineering principles and practices, such as model-driven design and Docker orchestration with Swarm can support the efficient

and productive deployment and management of a large number of Docker containers. The adoption of a distributed architecture can further enhance the efficiency and productivity of a small research group through the parallelization of tasks and the utilization of additional resources.

Overall, our findings indicate that the careful implementation of software patterns, microservices, Docker, and advanced software engineering principles and practices can significantly improve the reliability, scalability, performance, efficiency, and productivity of bioinformatics software systems. It is crucial to consider the needs and constraints of a bioinformatics software system and the availability of resources and expertise to optimize its success with these technologies. Future work could involve further exploration of the impact of these technologies on bioinformatics software systems, as well as the development of best practices for their implementation in this context.

3.5.3 Challenges and limitations

Throughout our research endeavour, we encountered a plethora of obstacles and limitations that required diligent attention in order to be effectively addressed. One challenge was the vast amount of tools and APIs that needed to be considered in order to make the system is versatile. Several tools incorporate microservice architecture. Three of the main tools are: Refget, Gen3 Framework Service, PhenoMeNal project [23, 60, 139, 143, 184]. Refget is a Global Alliance for Genomics and Health (GA4GH) API specification for accessing reference sequences and sub-sequences using an identifier derived from the sequence itself [143, 184]. The Gen3 Framework Services are a set of microservices that have been used to develop over a dozen different data commons and data resources, each developed independently for a different research community [23]. The PhenoMeNal project pioneered Virtual Research Environments (VREs) for metabolomics based on the microservice architecture, with many microservices in metabolomics analysis and general data analysis [60, 139]. There are further frameworks such as NCI Genomic Data Commons, refgenie and FAIRSCAPE with various focuses and take a leading role in biological computing [80, 108, 164, 165]. All of these projects are designed to facilitate the sharing of clinical and genomic data, and improve interoperability in the field of bioinformat-

ics. Building a software architecture that can handle and interact with other tools is a challenge that has to be outweighed and considered when designing the framework. Generally, the vast amount of tools highlights the importance of thoroughly evaluating and selecting software design decisions to ensure the functionality and interoperability of software architecture.

A limitation of our study was the limited scope of the thesis. The the topic of bioinformatics microservices is vast, and there are many additional points that could be considered. In particular, the research requires a deep understanding of software engineering principles such as Design Methodology for Reliable Software Systems [111, 133] and Designing and Deploying Internet Scale Services [79, 134]. On the bioinformatics side, there are also numerous software engineering topics that are essential for building a flexible data analysis system, such as data handling and tool integration for various analysis methods, as outlined in several research papers [89, 112, 122] . These topics are just a few examples of the many areas that need to be addressed when building software systems for bioinformatics. However, due to constraints on time and resources, we were unable to explore all fully aspects of this topic in depth.

Despite the challenges and limitations that we encountered during the research process, we were able to address them successfully and produce valuable insights into the use of software patterns, microservices, Docker, and advanced software engineering principles and practices in bioinformatics software systems. In addition to our theoretical analysis, we also implemented many of the concepts and principles that we studied, further demonstrating their feasibility and effectiveness in the context of bioinformatics software systems. Our research showcases the potential of these technologies in addressing the complexities and demands of bioinformatics software systems.

3.5.4 Future work

As we look towards the future potential of our bioinformatics microservices system, several exciting opportunities for improvement present themselves. As mentioned in our research, one promising direction for development is the expansion of the system's REST API to consume more tools and APIs. This expansion would significantly

increase the versatility and functionality of the system and enable it to interact with a broader range of resources and tools.

Another aspect to consider is the integration of advanced software design patterns and architectural styles to improve the reliability and maintainability of the system. By applying these principles, we can ensure that the system is robust and able to adapt to change requirements and demands, ultimately supporting the long-term success of the system.

The integration of new tools and resources could also be a valuable consideration, as it could enhance the capabilities of the system and enable it to address a broader range of needs in the bioinformatics community. The addition of a Common Workflow Language layer to the architecture may also help expand the API and enable interoperability with other systems.

Overall, we are confident that these strategies will allow us to continue improving the efficiency and effectiveness of our system and better serve the needs of the bioinformatics community. By embracing the latest technologies and principles in software engineering, we can support the advancement of scientific research and make a positive impact on the field of bioinformatics.

3.6 Conclusion

In this study, we sought to understand the influence of software patterns, Docker, microservices, and advanced software engineering principles and practices on reliability, scalability, performance, efficiency, and productivity of bioinformatics software systems. Through our research, we have identified several key factors that impact the success of these systems. The implementation of software patterns, such as the saga pattern and Command Query Responsibility Segregation (CQRS) pattern can enhance the system's ability to integrate data from multiple sources and improve scalability and reliability. The use of microservices and Docker containers can improve the modularity and maintainability of the system and facilitate the efficient deployment and scaling of the system. The adoption of advanced software engineering principles and practices, such as model-driven design and Docker orchestration, can further improve the efficiency and productivity of the system.

While our research has provided valuable insights into the design and implementation of bioinformatics software systems, we also acknowledged the challenges and limitations that must be considered. The vast number of tools and APIs available can make it difficult to evaluate and select the most appropriate technologies for a given system. Additionally, the scope of our study was limited, and there are many additional topics that could be explored in greater depth. Despite these challenges, we believe that our work has the potential to inform future research in this field and contribute to the development of best practices for the design and implementation of bioinformatics software systems.

Chapter 4

Enhancing Bioinformatics Analysis with Our Proposed System

Real-World Applications and Implications

The purpose of this chapter is to present the experimental results of our proposed system for biomedical software analysis. In order to provide context and understanding for the experiments, we will first review the critical software concepts highlighted in our study. Additionally, we will outline the background and core results that influenced the methods we used in our research. Finally, we will discuss how our results fit into the larger field of biomedical software analysis and contribute to the existing knowledge in this area. The results of this chapter are essential for advancing our understanding of biomedical software and its applications in the field with our proposed system.

4.1 Introduction

Software architecture plays a crucial role in designing and implementing bioinformatics systems. It defines the overall structure and organisation of the system, as well as the relationships between its components. Furthermore, a well-designed software architecture can support data and task integration, scalability, and parallelisation. This basis leads to improved performance and efficiency. However, designing software architecture for bioinformatics systems is a challenging task due to the complexity and heterogeneity of the data, as well as the need to support a wide range of analysis and management tasks. This chapter presents bioinformatics cases addressing these challenges through our proposed software architecture techniques. Our framework is designed to support the efficient analysis and management of large-scale biological data, and it utilises various techniques and technologies to achieve this goal.

In this chapter, we describe our experiments with this framework, which demonstrates its effectiveness in integrating data from multiple sources, scaling to handle large amounts of data and workload, and utilising parallelisation to improve performance. Our experiments show that the proposed framework can effectively integrate data from multiple sources and formats, even in the presence of data heterogeneity. It is also able to scale to handle increasing amounts of data and workload without a decrease in performance, thanks to its use of distributed computing infrastructure and cloud computing technologies. Furthermore, our experiments show that the proposed framework is able to utilise parallelisation to improve performance by

significantly dividing tasks into smaller subtasks that can be processed concurrently. Overall, our experiments demonstrate the effectiveness of the proposed framework in addressing the challenges of data integration, scalability, and parallelisation in bioinformatics.

The focus of this section is to investigate the impact of parallel computation and software techniques on the efficiency and accuracy of data analysis in bioinformatics. Specifically, the research questions are as follows:

- How does the implementation of parallel computation techniques in our framework improve the speed of data analysis in bioinformatics?
- What are the most effective approaches for integrating and analysing data from multiple sources within our framework in bioinformatics?
- How can our framework be scaled to handle effectively large-scale datasets in bioinformatics, and how can improve machine learning techniques are applied to optimise data integration and analysis in this context?

These questions aim to explore the potential benefits and challenges of using parallel computation and machine learning techniques in a bioinformatics framework to improve the efficiency and accuracy of data analysis. By addressing these questions, we hope to identify strategies for optimising the performance of our framework in the context of large and complex datasets and to develop approaches that can effectively integrate and analyse data from multiple sources.

4.2 Highlighted performance features

In this section, we will delve into the proposed system's performance characteristics through experimental trials. Specifically, we will evaluate three primary features: scalability, parallel distributed computing, and data integration. These performance features have been deemed crucial elements in the design and implementation of bioinformatics data analysis systems and thus have been selected as the focal point of our experimentation. The proposed system endeavours to address the challenges of scalability, parallel distributed computing and data integration to enhance the system's overall performance. Through a thorough analysis and review of relevant

literature, we will investigate the theoretical background of these performance features.

4.2.1 Scalability

Biomedical data analysis systems will continue to grow and expand, with an increase in both users and resources. This growth involves sustaining increased workloads as demand for the system increases [81]. The management of an increased workload is called scalability from a software development perspective. In section *2.4.1 Software engineering for biomedical data analysis*, we discussed scalability challenges such as monolithic architecture, which can make it difficult to adjust an existing system to meet new demands. In addition, scaling a software system is a complex process that requires careful consideration of time, complexity, requirements, and strategy. Therefore, it is essential to carefully assess these factors to successfully scale the system to meet the needs of a growing user base [81].

In biomedical data analysis, scalability is a key technical requirement that is reflected in the design goals of distributed computing systems [126]. Scalability challenges in these systems are outlined by Neuman in Section *2.2.2 Taxonomy*, and can be grouped into three main categories:

- Size: Challenges related to the growing number of requests and resources, such as services and data.
- Geography: Challenges related to the expanding distribution of users and computing nodes across different geographic locations.
- Administration: Challenges related to the increasing number of administrative organisations, such as research departments, that need to be taken into account.

These three dimensions of scalability deal with computing, network, and security/permissions issues.

To address the challenges of scaling systems, researchers have proposed several approaches, including replicating resources, utilising asynchronous network communication, and partitioning and distributing services [86, 172]. These approaches aim to address different aspects of the challenges.

Partitioning and distributing services can help to mitigate the challenges of increasing administrative organisations by dividing the system into independent environments, such as through virtualisation [86]. Asynchronous communication, on the other hand, provides a layer between a user and the service, allowing the system to compute resources rather than waiting for each task to complete [172]. Finally, replication creates multiple instances of a service to alleviate high demand on a single service [86]. Ultimately, these approaches offer solutions to specific challenges in scaling systems.

It is important to note that while each of these approaches can be effective on their own, they can also be combined for a more a comprehensive solution to scaling challenges. For example, replicating resources and utilising asynchronous communication can be combined to increase the efficiency of the system while partitioning and distributing services can be used in conjunction with replication to better manage increasing administrative organisations [172]. Overall, by carefully considering the specific challenges and selecting the appropriate combination of approaches, it is possible to effectively scale systems and overcome the various challenges that may arise.

The approaches mentioned above have interactive impacts on their respective fields of concern. These approaches may affect various Neumann scalability classes, depending on the perspective taken. Partitioning and distributing services can address increased administrative organisation and a growing number of services. From a software development operations perspective, partitioning the service into independent and enclosed environments allows for the distribution of implemented methods without compromising security or permissions. In other words, this viewpoint looks at the software system from a management/organisational viewpoint. Each service is an abstract entity.

The approaches mentioned above significantly impact their corresponding fields of focus. The effect on Neumann scalability classes may vary depending on the perspective taken. Partitioning and distributing services addresses the increased administrative organisation and the growing number of services. In the same vein, from a software development operations standpoint, the same pattern — namely, partitioning the service into independent and enclosed environments — enables the

distribution of implemented methods without compromising security or permissions. These two share standpoints to examine the software system from a management and organisational viewpoint. Therefore, treating each service as an abstract entity. However, when considering each service as an entity, we can partition the service into even smaller parts. From a microservice development perspective, partitioning the application into reusable and modular parts allows for managing a growing number of requests and resources. Therefore, the partitioning and distribution of the application can impact multiple Neumann classes, depending on the granularity of perspective.

This section will focus on the growing number of requests and resources dimension. As we elaborated on the other aspects in the section *3.2 Background: Technical Considerations for Bioinformatics Software Development* and *3.3.1 Microservice architecture*

Scaling a system can be complex, and various strategies have been developed to structure this process. As discussed in Section *2.2.2 Taxonomy*, horizontal and vertical scaling approaches can be used to increase the capacity of a system. While vertical scaling, or increasing the resources of a single machine, can be a fast solution, it has limitations. This limitation is exemplified by Amdahl's computation model, which states that "the overall performance improvement gained by optimising a single part of a system is limited by the fraction of time that the improved part is used" [142].

The theoretical speedup of a system is upper bounded based on Amdahl's formulation, as shown in Figure 4.1. In addition, increasing the fraction of time spent on the parallel portion can result in higher theoretical speedup than simply increasing the number of processors. Therefore, one key solution to scale a system is to scale horizontally or add more machines to the system to utilise resources better and improve overall performance. However, it is vital to consider the challenges associated with horizontal scaling, such as ensuring effective communication and coordination between the added machines. Therefore, it is crucial to carefully evaluate horizontal and vertical scaling options to determine the most effective system scaling strategy.

Given the significant advantages of horizontal scaling in terms of theoretical performance improvements, this study focuses on identifying and analysing software

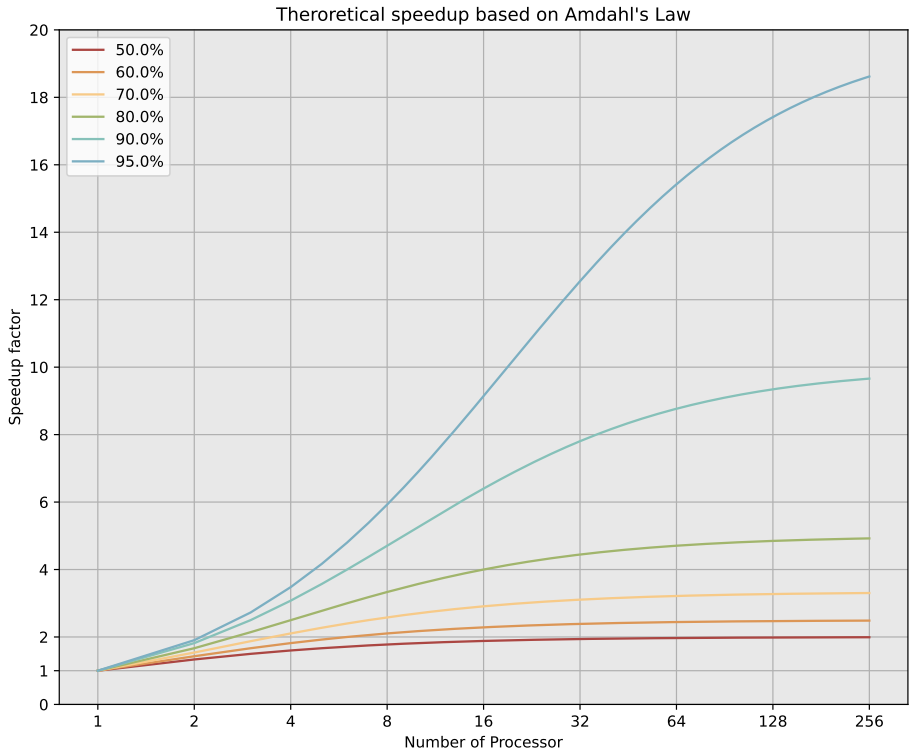


Figure 4.1: Amdahl's Law states that the maximum theoretical speedup of a program or task achieved by parallel processing is limited by the portion of the task that must be executed sequentially. This relation is depicted in the figure. The figure shows that the theoretical speedup also increases as the fraction of the task that can be parallelised (the parallel portion) increases. However, increasing the number of processors does not necessarily result in a higher theoretical speedup. Instead, the greatest improvement in performance comes from increasing the parallel portion of the task.

design patterns that facilitate the implementation of horizontal scaling techniques. Partitioning and replication are the two primary patterns employed in this context . Specifically, two patterns are discussed in detail: (1) partitioning and distribution and (2) replication. These patterns are systematically analysed concerning their applicability, benefits, and limitations to provide a comprehensive overview of their use in achieving theoretical speedup through horizontal scaling.

Partition and distribution

Partitioning and distribution are standard techniques used to design and implement complex systems. It involves breaking down the system into smaller, independent components, each assigned a specific function and located in a specific location. This approach allows for the optimisation of the system efficiency and organisation by distributing tasks and resources across multiple locations, rather than having all components residing in a single location. Partitioning and distributing a system makes it more scalable and resilient, as it can adapt to changing workloads and resources. Furthermore, this approach eases maintenance and modification of the system, as individual components can be updated or replaced without affecting the entire system. As a result, partitioning and distribution can significantly improve the performance and reliability of a system.

Problem Any software system has to handle general system failures. A variety of facets can cause system failures. In this instance, we are considering the software side of a system failure. For instance, a service or component is unable to process a request. In a centralised system, an unprocessed request can lead to a delay, inconsistent state, or a complete system crash. Therefore, any system design has to handle general software-based system failures. The main problem in a centralised system is that an inconsistent state of requests can affect the whole system.

In addition, only increasing resources in a centralised system could be more efficient than restructuring the system. As mentioned above, the theoretical speedup for the system, which uses a parallelisable portion of $\frac{1}{2}$, only doubles when increasing the processors by a factor of 16. If we compare this theoretical speedup to increasing the parallelisable portion, we can see that it is far from efficient. To achieve the same

speedup, we only need to increase the parallelisable portion to $\frac{3}{5}$ and the processors by a factor of 4. Hence, in a centralised system, increasing resources is inefficient and costly to improve speedup.

Solution and Features The software pattern of partitioning and distribution can solve the challenges mentioned earlier. By partitioning a system, we can achieve two benefits: first, dividing a monolithic system into service data processing creates independent paths so that a single concern only affects the related areas instead of blocking or affecting other parts. Therefore, since the system relies on several supports, it decreases the likelihood of a system crash. Second, in the process of partitioning systems in architectural design, improving the parallelizability fraction improves theoretical speed. Then, distributing the services can also improve response time by bringing the service closer to the resources. All in all, partitioning and distributing a system into services allows for the following:

1. a decrease in the probability of system crashes
2. an improvement in theoretical speedup
3. a decrease in response time

Technical layaout When designing a system, various design patterns can be applied to different levels, as illustrated by Krutchen in the section *3.2 Background: Technical Considerations for Bioinformatics Software Development.* This illustration helps structure the processing system and consists of four levels: logical, development, process, and physical.

As outlined in the development level of Krutchen's illustration, we employed a model-driven design utilising microservices. This approach is discussed in more detail in sections *3.3.1 Microservice architecture* and *3.4 System Implementation.* By separating concerns, we can minimise service interference, with the worst-case scenario being that only a single service would be affected by a system inconsistency rather than the entire system. From a physical perspective, we utilised containerised virtualisation to provide an additional layer of separation for our services. This design also allows for the distribution of services based on requests, similar to a

DNS resolving method. As a result, this distribution helps reduce the time for data transfer, as the processing and data are closer to the request.

We employed a combination of two partitioning techniques to address scalability challenges: model-driven microservices and containerised virtualisation. We applied these patterns at the system development and physical levels to minimise the probability of system failure and reduce response time.

Replication

Problem As mentioned above, having a single system with a centralised and single instance can bring challenges. First, a single process request in a sequential manner can block other threads, causing the entire system to wait until the request is finished. Additionally, relying on a single instance of a system reduces its stability. Furthermore, after a system failure, all processes, even unrelated ones, are cancelled and aborted in the worst case. A restart might cause an inconsistent state, so the system cannot restore the previous state. Therefore, a single-process centralised system blocks other processes, increases stability vulnerability and can cause data inconsistencies.

Solution and Features These challenges can be resolved with the software design pattern of replication. Replication creates copies of resource and service instances. The system can be more stable with multiple service instances since several instances can serve resources to a request. Moreover, depending on the replication implementation, work can be handled by a set of nodes instead of sequentially. Finally, replicating resources and keeping a data copy close to the request reduces the time needed to request data. Therefore, the replication pattern increases stability, provides parallel execution, and decreases data loading.

Functionality We use two common approaches to replication: service separation, parallelisation, and caching. Regarding service separation, we use standard software decomposition methods. In our implementation, the replication pattern creates several instances for computing. This implementation includes instances for the UI, general logic, and task execution. Furthermore, our implementation includes a con-

trolling node, which organises tasks for the worker. Specifically, the controlling node persists the metadata for every task so that each worker only receives the necessary data needed to perform a task. Additionally, we implemented a task partitioning method to break up more significant tasks. From a caching perspective, each worker pulls the required data from the defined source. Each worker has a copy of the original data, improving data consistency. Therefore, we used a general software pattern of software decomposition with instance replication and a controlling node to implement the replication pattern.

4.2.2 Distributed system parallelisation

Distributed systems have several advantages. One of the main advantages is parallelisation. In Distributed computing, parallelisation can refer to a set of software patterns. Each pattern solves a particular problem in data-driven software development. Therefore, we focus in this section on data and task parallelism.

As mentioned in the section *1.1 Scope and Challenges* and *2.4 Open issues and research challenges in biomedical data analysis*, some of the core challenges in data-driven software analysis is long runtime application executions due to sequential task executions, large data sets, and inferring new results from data. These challenges are especially apparent in biomedical data analysis. In biomedical data analysis, major computational methods rely on fundamental transformations on vectors, matrices, or tensors. These linear algebraic transformations benefit from parallel execution in several ways. First, the time for a sequential task can be reduced. Second, large data sets can be handled by a group of nodes. Third, parallelisation allows for simultaneously executing data analysis from several perspectives. Therefore, parallelisation in a distributed system is a core advantage in biomedical data analysis.

When integrating any form of software-level parallelisation into a system, developers must consider many factors. Two main factors are software topology and parallelisation mode. These two aspects are closely related and affect each other mutually. Therefore, software architecture decisions impact the degree of parallelisation.

In parallel computing, the term topology refers to the underlying software architecture. We used a distributed software architecture. The core aspects of a dis-

tributed service-oriented architecture are component concurrency, individual system clocks, and independent components. These aspects have several benefits, such as implementing parallelisation for biomedical data-driven analysis. Therefore, with a distributed software architecture, the parallelisation mode can be built on a flexible infrastructure.

We considered two main approaches for the parallelisation mode: task and data parallelisation. As mentioned above, the core advantage includes concurrent task execution, which reduces the time compared to sequential execution. The two parallelisation approaches, task and data, handle concurrent execution differently. We outline the key concepts on a general level. Task parallelisation is a parallelisation mode across several processors or nodes. Generally, Task parallelisation is part of parallel computing environments. However, task parallelisation can also be applied to distributed systems. Task parallelism focuses on distributing tasks concurrently. Depending on the implementation, distributing concurrent tasks performs the task by a set of processors, nodes or threads across a system. For instance, a single data set moves through separate tasks in pipelining. Each task can run independently from the other. In contrast to task parallelisation, data parallelisation involves performing the same operation on different data. Generally, data parallelism involves dividing the same task among different data components. Each processor or node performs the same task on different distributed data. This process is generally part of a multiprocessor system in a single set of instructions [34, 50, 174]. An example where data parallelism can improve runtime is matrix multiplication. Consider the matrix $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$. For the multiplication $A \cdot B$, each entry in the result C can be calculated individually. This partition would result in a run time of $\mathcal{O}(m \cdot n \cdot p)$. By running the task parallel, the runtime can be reduced. The multiplication can mainly be completed in $\mathcal{O}(n)$ when executed in parallel using $m \cdot k$ processors. We illustrated the task and data parallelisation modes in figure 4.2. Therefore, the parallelisation modes, tasks and data mainly differ in executing concurrent tasks by splitting the task or the data.

By implementing task parallelisation, a set of processors, nodes or threads can perform a single task simultaneously, which can help to reduce the execution time of the task when compared to the sequential execution. On the other hand, data

parallelisation, the same task is performed on different data by different processors, nodes or threads, which can also help to speed up the overall process.

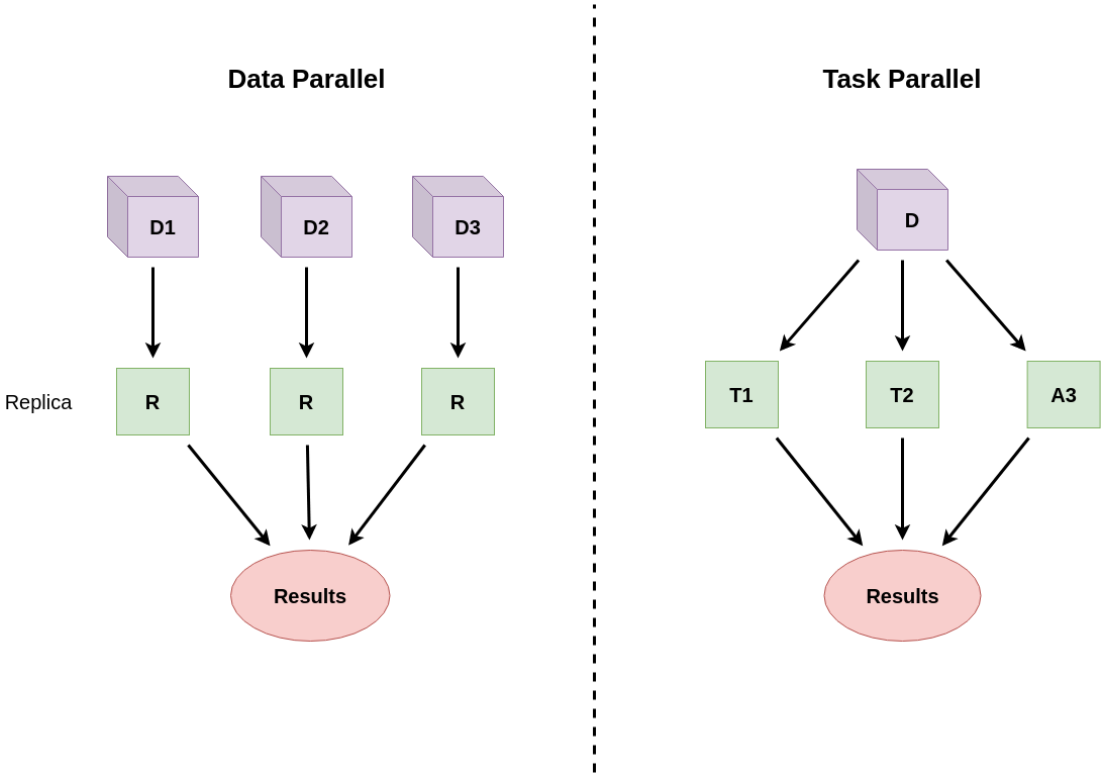


Figure 4.2: Data parallelisation and task parallelisation are distinct strategies for parallelisation, each possessing unique characteristics and areas of applicability. Data parallelisation involves partitioning data into smaller subsets and the concurrent execution of the same task on each subset. On the other hand, task parallelisation involves assigning different tasks to each subset and concurrently processing the subsets. These approaches to parallelisation have been widely utilised to improve the efficiency and speed of data processing operations.

4.2.3 Data integration

Integrating bioinformatics data from multiple sources can be complex due to the diversity and lack of standardisation in the data collection, storage, and annotation process. The heterogeneous data sources may use different data formats, schemas, and terminologies, making it challenging to integrate the data in a meaningful way.

Moreover, scalability, data security and privacy, and data integrity are essential concerns that must be addressed while integrating large amounts of bioinformatics data.

our proposed system addresses these challenges by utilising Docker Swarm, a representational state transfer-based application programming interface, and object storage as crucial components. Docker Swarm is an orchestration and management tool for clusters of Docker containers, which allows for efficient scaling of the application by adding more containers to the cluster as needed, which can be helpful when integrating large amounts of data from multiple sources. Additionally, using Docker containers can ensure the security and privacy of the data by isolating and protecting the application and its data from the host environment.

The system also implements a REST-based API, which provides a consistent and predictable interface for accessing and integrating data from different sources. Representational State Transfer (REST) is a software architectural style that defines a set of constraints for creating web APIs. By exposing a REST-based API, the system can address issues related to the heterogeneity of data sources and lack of standardisation, as it provides a common way to access and manipulate the data.

Object storage is also utilised as a part of the system, which can help address some of the problems associated with data integration, particularly those related to scalability, data security, and privacy. Object storage is a type of data storage designed to store and retrieve large amounts of unstructured data, such as files and images. It is typically highly scalable and can handle large volumes of data, making it well-suited for storing and managing large amounts of data in a data integration system. Additionally, object storage systems often include built-in security features, such as encryption and access controls, which can help ensure the security and privacy of the data.

Overall, the utilisation of Docker Swarm, REST-based API, and object storage, make this data integration framework a promising approach for addressing challenges in the field of bioinformatics.

In the upcoming section, we will delve into the three performance features of scalability, distributed parallelisation, and data integration. Our research will evaluate the effectiveness of these features in our system and how they work together

to optimise performance. We will conduct experiments to assess the integration of these features within our system and analyse the results to conclude their impact on overall system performance. This analysis will provide valuable insights into the importance of these features in modern systems and how they can be utilised to achieve optimal performance.

4.3 Biomedical experiments

In this section, we conduct experiments on our system to evaluate the performance characteristics of the proposed approach. We provide a detailed description of the experimental setup for each experiment, including the specific embedding techniques used in bioinformatics and mathematics. Additionally, we present and discuss the results of each experiment, highlighting how they demonstrate the performance features of the proposed system.

4.3.1 Matrix decomposition on Genotype-Tissue Expression Project data

The term "omics" in biology refers to a set of disciplines that study large-scale biological systems, such as genomics, proteomics, and metabolomics. Consequently, omics data can contain signal interactions in the form of kinetic, physical or molecular interactions that control biological systems.

Generally, omics data is stored in data structures resembling vectors, matrices, or tensors. With new data-generating possibilities, large data sets have grown, leading to an increasing data dimension. Moreover, with increasing data size, applying a set of sequential analyses can result in long computing times. Therefore, we require a method and infrastructure to handle large-dimensional omics data sets.

One method that can handle these immense data challenges is matrix decomposition. Matrix decomposition is a technique that reveals low-dimensional structure from high-dimensional data. In omics data, the low-dimensional representation can outline signal interactions. matrix decomposition is widely used in omics data and has proven to be effective in revealing new biological knowledge from various omics

data [62, 163]. Furthermore, we can reduce computational time by using an appropriate parallel software system. Hence, matrix decomposition combined with a parallel software system provides opportunities to infer relevant features from data and reduce computing times.

In this section, we will review the data structure of omics data, e.g. genomics, transcriptomics and proteomics. First, we will outline the general data structure for omics data. Afterwards, we will describe the biological representations and implications of matrices and discuss how they can be used to model biological systems and extract meaningful information e.g. gene expression in different human tissues. Then, we will mathematically outline three analysis methods: principal component analysis, independent component analysis, and non-negative matrix factorisation. Finally, we will present our experimental results using matrix decomposition on RNA-seq data from a GTEx dataset. We will demonstrate how the technique can be applied to this specific dataset and what insights it can gain.

Matrix decomposition with omnics data

In biology, new advancements in experimentation unfold a new era in data-driven analysis. These novel experiments employ various data aggregating methods, such as data processing/control software, liquid handling devices, and sensitive detectors. This aggregated data serves as a starting point for numerous tests. These tests may include several omics fields, i.e. genomics, proteomics, metabolomics, metagenomics, phenomics and transcriptomics. For instance, in genetics, several tests allow the classification of observable traits or characteristics of an organism, i.e. phenotypes. Concretely, in genetics, two main aspects characterize phenotypes. First, proteins interact with transcription factors to activate or repress the transcription of specific genes. Second, multiple molecules and processes influence the cooperatively and synergistically on the phenotypes. These two effects are part of a general group of complex biological process. Apart from highly specialized genetic tests, the tests on extensive aggregated data are also referred to as high-throughput screening. Consequently, in the field of genetics, high-throughput screening allows a more detailed classification of complex biological process.

Running tests on this aggregate data requires data structures containing all nec-

essary data. In most cases, the commonly used data structure are matrices. These matrices can hold several values, e.g. expression counts, methylation levels, and protein concentrations. Commonly the matrix rows contain genes. Whereas each column represents an individual/patient sample. We illustrated a general analysis pipeline overview in figure 4.3. One of the core tasks in finding structures in these large matrices is to classify phenotypes or complex biological process. Therefore, we need methods to discover phenotypes and complex biological process from biological data in matrixes.

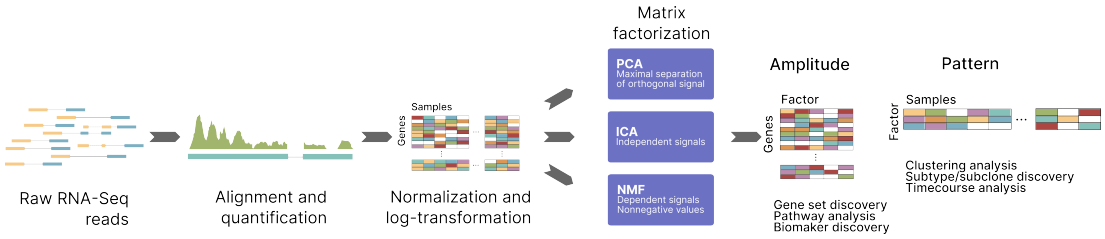


Figure 4.3: The analysis pipeline consists of several stages, including preprocessing raw RNA-sequence reads, alignment and quantification, normalization and log transformation, matrix factorization using PCA, ICA, and NMF, and matrix decomposition. In this study, we concentrate on the matrix decomposition step in the pipeline.

From a genetics viewpoint, the representation as a matrix has a significant implication for further analysis [33, 116]. The relationship between the two aspects allows a simplified representation of extensive data. These core aspects are complex biological processand similar phenotype samples. Only a tiny subset of complex biological processis active at a given time. Consequently, a good analysis technique can extract low-dimensional from high-dimensional data. The commonly used technique is matrix decomposition. Matrix decomposition — also referred to as matrix factorisation— is a decomposition method that factors matrix into a product of matrices. Thus, the method matrix decomposition uncovers underlying structures in genetics by specifying the genetic context to input data in the form of matrices.

Generally, matrix decomposition provides a simplified representation of a matrix. As we only consider genetic data, we will consider matrices over $\mathbb{R}$ — namely $\mathbb{R}^{n\times m}$, with $n, m \in \mathbb{N}$. Numerically, we want to approximate or represent a matrix

as a product of two or more matrices:

$$X \approx A \cdot B \tag{4.1}$$

where $X \in \mathbb{R}^{n \times m}, A \in \mathbb{R}^{n \times p}, B \in \mathbb{R}^{p \times m}$. This problem is equivalent to

$$A \approx X \cdot B^{-1} \tag{4.2}$$

For a detailed explanation of the existence and approaches behind this equivalenz consider *A.1.1 Inverses*

In conclusion, new advancements in data-driven analysis in biology have led to the use of various data aggregating methods and the application of these methods to various omics fields. These tests allow for a more detailed classification of cellular and biological processes. Data representation in matrices allows for a simplified representation of extensive data. It facilitates further analysis using techniques such as matrix decomposition, which uncovers underlying genetic structure by specifying the genetic context to input data. The representation of data in matrices also makes it possible to approximate or represent a matrix as a product of two or more matrices, providing a simplified representation of the data. With the increasing amount of genetic data being generated, techniques such as matrix decomposition will continue to play a critical role in analysing this data.

Matrix decomposition methods

In this section, we will provide an overview of the most commonly used matrix decomposition methods, namely Principal Component Analysis (PCA), Independent Component Analysis (ICA), and Non-negative Matrix Factorization (NMF). These methods, known collectively as matrix decomposition, vary in their approach to calculating each factor. We will outline the core principles of each approach with mathematical aspects that allow for the implementation of matrix decomposition techniques. By understanding the fundamental concepts behind each technique, we can better utilize them to extract meaningful features and representations of our data.

Principal component analysis Principal Component Analysis (PCA) is a widely-used technique for dimensionality reduction in bioinformatics. It is a mathematical method that utilizes linear algebra and statistics to identify patterns and relationships within high-dimensional data. Identifying and removing less important features allows for the visualization and interpretation of complex data in a lower-dimensional space.

PCA is based on the concept of eigenvectors and eigenvalues, which are derived from the covariance matrix of the data. The eigenvectors with the highest eigenvalues are chosen as the principal components, which are linear combinations of the original features that capture the most variation in the data. These principal components can then represent the original data as new, uncorrelated features. .

In bioinformatics, PCA has been used in many applications, such as gene expression analysis, image analysis, and protein structure analysis. It has also been utilized as a preprocessing step for other data mining algorithms, making it is a powerful tool for data exploration and discovery in bioinformatics research.

The matrix decomposition method principal component analysis is one of the most commonly used approaches for dimension reduction using linear algebraic and statistical methods. Generally, principal component analysis recognizes geometric similarities by using algebraic properties. We will outline the core linear algebra and statistical results that allow principal component analysis.

Independent component analysis The matrix decomposition approach, independent component analysis, is utilized to address the challenge outlined in section *4.3.1 matrix decomposition with omnics data* through the use of equations 4.1 and 4.2 as described in previous literature [40, 84]. Independent component analysis is further enhanced by incorporating an additional layer of probability theory to model the method's context. The detailed mathematical results are expanded in *A.1.3 Independent component analysis*.

In comparison to Principal Component Analysis (PCA), ICA focuses on maximizing the statistical independence among the resulting components rather than variance in the data. The technique is founded on the principles of non-gaussianity and the assumption that the underlying signals are independent and non-gaussian. Higher-order statistics such as kurtosis and entropy are employed to identify the inde-

pendent components by maximizing the non-gaussianity of the resulting components as described in [40, 84].

The foundation of ICA is based on the mathematical formulation of the problem and the utilization of probability theory to model the underlying signals. The central limit theorem, the law of large numbers, and maximum likelihood estimation are among the mathematical theorems employed in ICA. This technique has been utilized in various bioinformatics applications such as gene expression analysis, image analysis, and signal processing [183].

As a critical remark, the resulting components generated by independent component analysismay not be orthogonal to each other, in contrast to those generated by principal component analysis. Furthermore, independent component analysismay be computationally more demanding than principal component analysis; however, it is particularly practical when there is an assumption of non-gaussian independence among underlying signals.

Non-negative Matrix Factorization (NMF) The matrix factorization approach, NMF, is used to extract essential features in a dataset and to represent it in a more interpretable and low-dimensional space. NMF assumes that the data is composed of non-negative components. This fact is used to construct the factorization [103,104]. The algorithm aims to approximate the data matrix as the product of two non-negative matrices, usually the basis and the coefficient matrix. The detailed mathematical results are expanded in *A.1.4 Non-negative matrix factorisation.*

In bioinformatics, NMF has been applied to a wide range of problems, such as gene expression analysis, text mining, and image analysis. In gene expression analysis, for example, NMF can be used to identify interpretable patterns and representations in gene expression data, it has been widely used for clustering and dimensionality reduction [76, 109]. NMF has also been used in image processing and computer vision, where it can extract features such as edges and textures from images, by using non-negativity constraints, it provides a more interpretable representation of the image [102, 103].

NMF has some advantages over other methods like PCA and ICA. NMF's non-negativity constraint makes the results more interpretable and easy to visualize, and it is robust to noise and missing data and can cope with sparse data as well. Addi-

tionally, the optimization problem of NMF can be relaxed to a convex optimization problem, making it computationally more efficient and simpler to implement [53,54].

In summary, Non-Negative Matrix Factorization (NMF) is a powerful and widely-used method in bioinformatics. It allows extracting interpretable features from high-dimensional data, and it is based on the non-negativity the constraint of the original data. NMF has been applied in a wide range of bioinformatics problems, such as gene expression analysis, text mining and image analysis and its computational efficiency and robustness to make it a suitable choice for real-world datasets.

In conclusion, matrix decomposition methods such as PCA, ICA, and NMF are widely used in bioinformatics to extract essential features and representations of data. These methods have been applied to various problems, such as gene expression analysis, image analysis, and signal processing. Understanding these techniques, principles, and mathematical foundations can help researchers choose the most appropriate method for a specific problem. It is crucial to consider the properties of the data, the assumptions of the model, and the computational complexity when selecting a matrix decomposition method. We can gain valuable insights from large and complex data sets in bioinformatics with a deeper understanding of these techniques.

Matrix decomposition Experiments

As mentioned in section *1.1 Scope and Challenges* , one of the main challenges in biomedical data analysis is to manage large amounts of data and extract information from a data set. We choose matrix decomposition approaches to resolve these challenges in biomedical data analysis. Namely, we used different matrix decomposition approaches on one dataset in parallel, as mentioned in section *4.3.1 matrix decomposition with omnics data*. With our approach, we can reveal various aspects dataset. Furthermore, we can evaluate the different methods —principal component analysis, independent component analysis, and non-negative matrix factorisation— on how they perform. Therefore, by using task parallelisation computing strategy with several matrix decomposition approaches, we can highlight the active research field to uncover hidden information in a data set.

We execute matrix decomposition methods to outline the different approaches: principal component analysis, independent component analysisand non-negative ma-

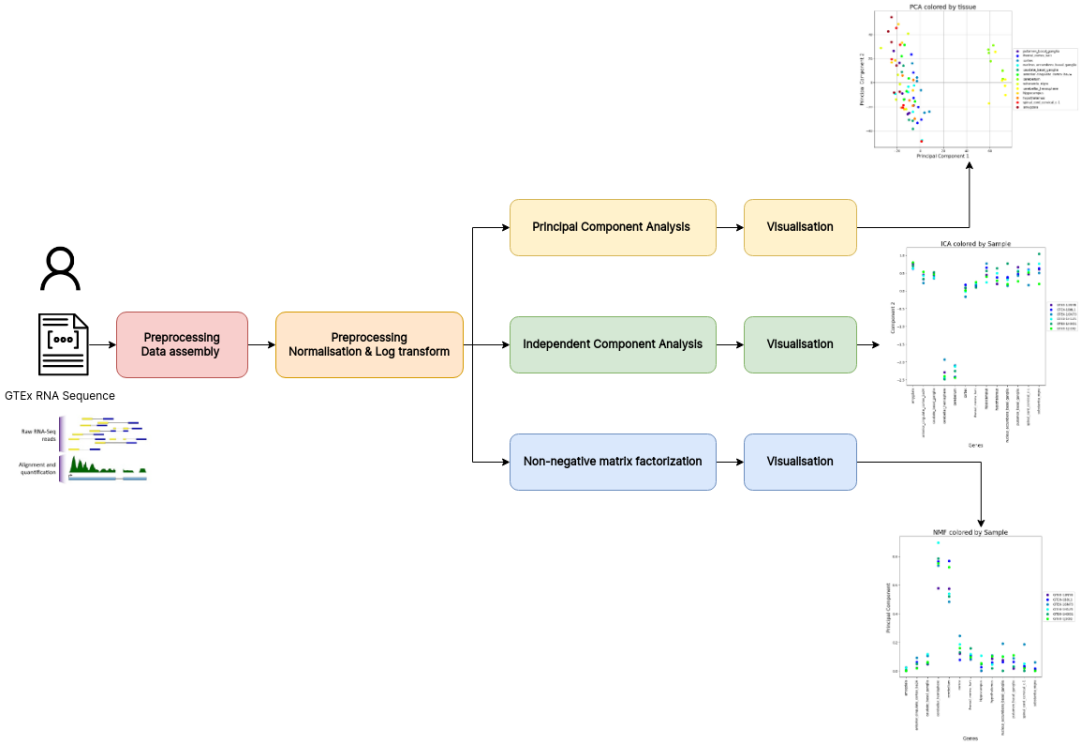
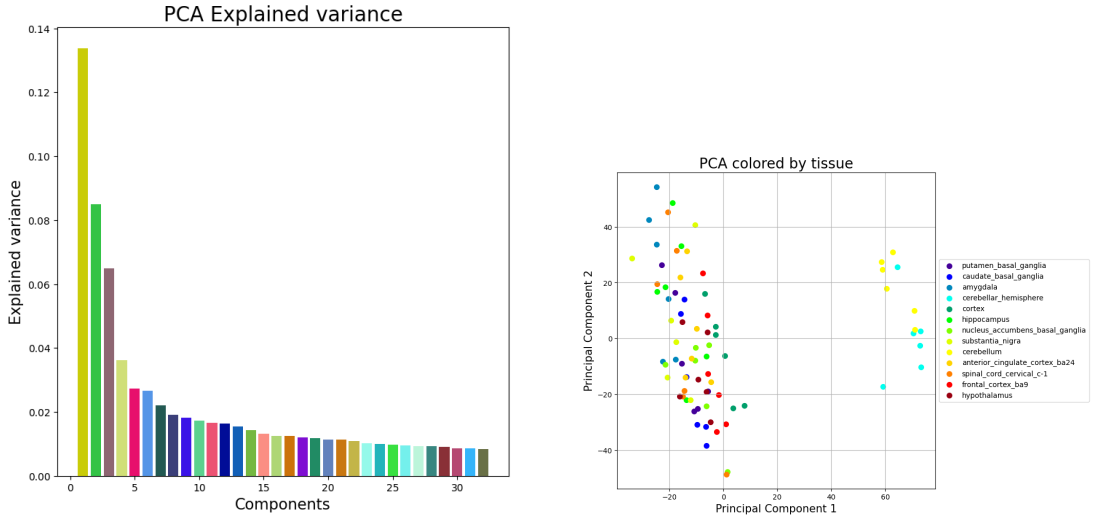


Figure 4.4: We focused on the analysis of GTEx RNA sequences using a custom-designed analysis pipeline. The pipeline included several stages, beginning with assembling data from raw sources. Following data assembly, we applied to preprocess techniques such as normalization and log transformation to the data. To facilitate efficient and concurrent execution, we implemented a distributed execution strategy which allowed for the intended parallelization of the analysis.

trix factorisation. We applied the methods to a single GTEx dataset [52]. This dataset consists of 13 brain tissue postmortem samples for six individual patients. This experiment condition has a major benefit: the complex biological process are labelled, which allows a basis for a matrix decomposition method comparison.

First, we executed principal component analysis to the GTEx dataset. In order to start the analysis, one essential aspect can reveal further insights: explained variance. Explained variance refers to the amount of variability in a dataset. This variability relates to each principal component. Generally, explained variance refers to the proportion of variability in a dataset that can be attributed to each principal component. It gives an idea of how much each component contributes to the total



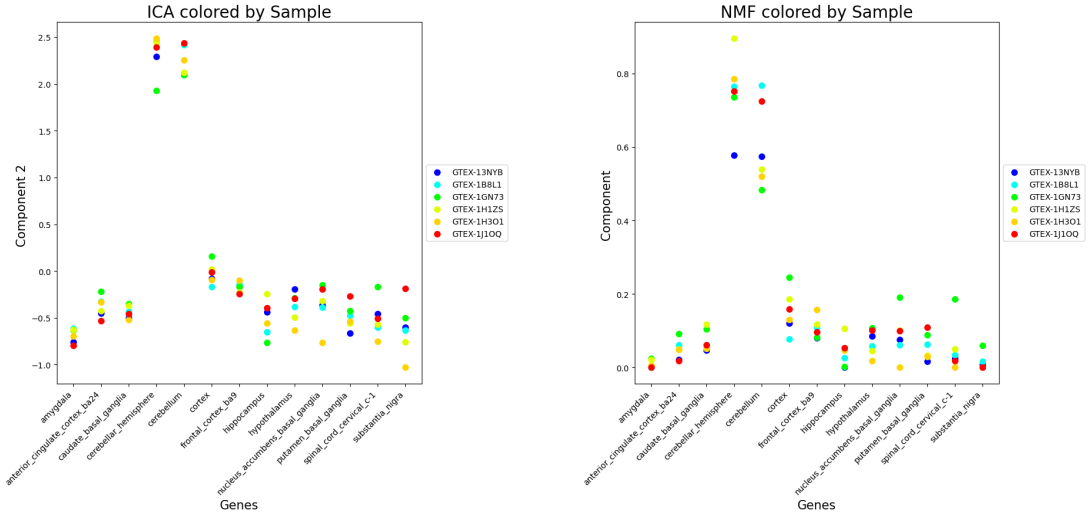
(a) Explained variance **(b)** Principal component analysis

Figure 4.5: The figure illustrates the results of applying principal component analysisto the data. Figure (a) illustrates the explained variance of the data when using principal component analysis. This means, in our case, that the differences and variations observed within the dataset can be primarily attributed to these three components rather than other factors. Figure (b) depicts a scatter plot generated using PCA. The x and y axes represent the two principal components of the data. Each point corresponds to a single observation, showing the variable values for that observation on the x and y axes. It visualizes the relationship between the different variables in the data. The plot demonstrated the separation of the cerebellum and cerebellar hemisphere, suggesting that these two tissues can be distinguished from each other

variability in the data. In our experiments, the first components explain about 22% of variation in this data as illustrated in figure 4.5a. Even though the amount is relatively small, all the other components contribute less than 2% to the total variance. Generally, principal component analysis inspects the relation between these components from a biological viewpoint using low-dimension plots as illustrated in figure 4.5b. This approach this similar to clustering methods since these plots can separate biological features in the data. In our experiments, we observed that the cerebellum (represented in yellow) and the cerebellar hemisphere (represented in

cyan) were distinguished from the other brain tissues in the data, as revealed by the low-dimensional plots.

As mentioned in an earlier section concerning independent component analysis and non-negative matrix factorisation the components are independent. Therefore, we plotted one component concerning the brain tissue. Each plot distinguished the cerebellum and the cerebellar hemisphere when we applied the independent component analysis and non-negative matrix factorisation to the GTEx dataset. These two brain tissues have relatively large absolute values compared to the other brain tissues. Therefore, we can infer that the independent component analysis and non-negative matrix factorisation facilitate a tissue-specific classification, whereas principal component analysis facilitates a tissue segregation approach. Concretely, independent component analysis may group complex biological process with common genes where a value sign changes due to experimental conditions and gene under- and overexpression. On the other hand, non-negative matrix factorisation reveals tissues that are only overexpression of genes.



(a) Independent component analysis **(b)** Non-negative matrix factorisation

Figure 4.6: We used independent component analysisand non-negative matrix factorisationto analyze the tissue samples. We plotted the tissue samples against one component for both methods to visualize the results. The plots clearly showed separation between the cerebellum and cerebellar hemisphere, indicating that these two tissues can be distinguished from one another based on this component.

Each matrix decomposition approach observes particular features inside the data set. Generally, we can infer different aspects — e.g. various complex biological process or phenotypes — within a single dataset. We reveal new insights with various approaches: Using different matrix decomposition methods or changing parameters on method execution. Our matrix decomposition experiments are standard analysis methods in biomedical data analysis. These experiments can be expanded to highlight more details on a single dataset. Nevertheless, the general pipeline is similar to our approach. All in all, we can summarise our findings for matrix decomposition experiments in the GTEx as follows:

1. principal component analysis allows separating different brain tissues in the sample
2. independent component analysis locates under- and overexpression of genes
3. non-negative matrix factorisation only finds overexpression of genes

Even though independent component analysis might include the findings of a non-negative matrix factorisation, one the drawback might be a large mixture of various complex biological process. Therefore, depending on the analysis approach, each of the matrix decomposition provides a spectrum of analysis methods essential for exploratory biomedical data analysis.

4.3.2 Hidden Markov model search on prototypic sequences representing repetitive DNA from different eukaryotic species

In the field of bioinformatics, one of the critical challenges is the classification of protein families and the identification of homologous sequences in protein databases. This challenge is a crucial task, as it allows researchers to understand the structure and function better of proteins, which is crucial for a wide range of applications, including drug discovery and the development of new medical treatments.

One approach that has been widely used for this task is the use of Hidden Markov Models (HMM). HMMs are probabilistic models that can represent the sequence of

amino acids in a protein, considering the insertion and deletion of amino acids and the emission probability of each amino acid. Using these models, researchers can search large protein databases to identify sequences similar to a target sequence.

The use of HMM for protein sequence analysis has proven to be a a powerful tool, allowing researchers to quickly and accurately classify protein families and identify homologous sequences in protein databases. In this section, we describe our approach to using HMM for protein sequence analysis, including the mathematical foundations of HMM, the steps involved in the analysis workflow, and the benchmark results of our approach. By providing a detailed overview of our approach, we hope to contribute to the ongoing efforts to better understand the structure and function of proteins.

Repbase is a database of repetitive elements in eukaryotic genomes. Repetitive elements are stretches of DNA present in multiple copies within the genome and can account for a significant portion of the genome. Repbase is often used in bioinformatics studies to identify and annotate repetitive elements in genomic sequences. It can also identify transposable elements, a repetitive elements that can move or "transpose" within the genome. Repbase is a valuable resource for researchers studying genome structure and function.

Hidden Markov model in bioinformatics

A Hidden Markov Model (HMM) is a mathematical model used in bioinformatics and other fields to analyse sequences of observations [57]. It is a statistical model that assumes that the underlying process that generated the observations is a Markov process with unobserved (i.e., hidden) state variables.

In bioinformatics, HMMs are often used to analyse protein and nucleotide sequences. The model consists of states, each corresponding to a particular sequence pattern. The states are connected by transitions, representing the likelihood of moving from one state to another. The emission probabilities of the model determine the probability of observing a specific symbol (e.g., an amino acid or nucleotide) in a given state.

Mathematically, an HMM can be defined as

Definition 4.3.1. *A Hidden Markov model is a tuple* (Q, Σ, A, π)*, where:*

- $Q = q_1, q_2, ..., q_n$ *is a set of states*
- $\Sigma = x_1, x_2, ..., x_m$ *is the alphabet of symbols that can be observed*
- $A = (a_{ij})1 \leq i, j \leq n$ *is the state transition matrix, where* a_{ij} *is the probability of transitioning from state* q_i *to state* q_j
- $\pi = (\pi_1, \pi_2, ..., \pi_n)$ *is the initial state distribution, where* π_i *is the probability of starting in state* q_i*.*

Given an HMM and a sequence of observations, the goal is to infer the most likely sequence of hidden states that generated the observations. This goal can be achieved using the Viterbi algorithm, which computes the maximum likelihood estimate of the hidden state sequence.

For example, consider an HMM used to model the sequence of nucleotides in a DNA strand. The hidden states in this HMM might represent the different possible secondary structures of the DNA, such as helices or loops. The observations would then be the individual nucleotides in the sequence, and the emission probabilities would describe the likelihood of emitting a specific nucleotide from each hidden state. Using HMM algorithms, one can then use this model to identify the most likely sequence of hidden states given an observed sequence of nucleotides [57, 99].

One of the main advantages of using HMM search in bioinformatics is that it allows for incorporating uncertainty and variability into the analysis of protein or nucleic acid sequences. For example, a protein may have multiple possible states at a given position, and HMM search can account for this uncertainty by calculating the probabilities of different states occurring at each position. This approach can provide a more accurate and detailed analysis of the protein or nucleic acid sequence.

In addition to identifying the sequence of a protein or nucleic acid, HMM search can also be utilised for the analysis of the structure of the molecule. It can also be used to analyse the function of the molecule. For example, examining the transitions between different states makes it possible to identify functional domains or motifs within the protein or nucleic acid. This information can be used to understand the biological role of the molecule better and to design experiments or therapies that target specific regions of the molecule. Overall, HMM search is a powerful and widely-used tool in bioinformatics, and it continues to be an active area of research and development.

Protein families can be classified using HMM search. This classification involves converting protein sequence alignment into position-specific scoring systems called HMMs. For protein-specific HMMs, the insertion and deletion of amino acids determine the emission probability. HMMs can search through protein databases for homologous sequences using a scoring value of S. The scoring is defined as the log of the ratio of the joint probability of the target t. In addition, the alignment π given the homology H and the probability of the target given random noise R.

$$S = \log\left(\frac{\sum_{\pi} P(t, \pi|H)}{P(t|R)}\right)$$

In order to understand how HMM search works, it is vital to first understand the concept of protein families and homology. Protein families are groups of proteins that have similar sequences and, furthermore, likely have similar functions due to their similar structures. Homology, on the other hand, refers to the similarity between evolutionarily related proteins. Using HMMs to search through protein databases makes it possible to identify proteins that are homologous to a given protein of interest.

The use of HMMs in protein family classification and homology search has several advantages. One advantage is that HMMs can account for insertions and deletions of amino acids, which is vital for accurately aligning protein sequences. Additionally, HMMs can provide a probabilistic model for the alignment of protein sequences, which allows for a more robust and accurate analysis. Finally, the use of HMMs in protein family classification and homology search allows for the efficient search of large protein databases, making it a valuable tool for protein research.

To conclude, HMM search is a powerful tool for classifying protein families and identifying homologous proteins. Researchers can use position-specific scoring systems known as HMMs to accurately align protein sequences and search for homologous proteins in large databases. For more information on HMM search and its applications in protein research, interested readers can refer to Finn's work on the topic [64].

Hidden Markov model search pipeline

In this study, we propose a Hidden Markov Model (HMM) analysis based on the workflow developed by Ustyantsev et al. [170]. This workflow requires two main inputs: a Multiple Sequence Alignment and a sequence database [20]. The HMM analysis involves several steps, including the translation of DNA to protein sequences, HMM search, filtering of search results, and visualisation of results as illustrated in figure A.6. This study only focuses on the HMM search to assess the computational times.

To run an HMM search through a RESTful API, we need to deploy the service as described in section Technical Setup Steps. The algorithm scripts are contained in docker images that can be deployed on any machine with access to the system.

The Multiple Sequence Alignment presented in figure A.7 is a crucial input for the HMM analysis. It provides information about the sequence and structure of the proteins being studied. The sequence database [20] is also an essential input, as it contains a collection of known sequences that can be used for comparison during the HMM search.

The first step in the HMM analysis is to translate the DNA sequences into protein sequences. This step is necessary because the HMM algorithm operates on protein sequences rather than DNA sequences. The translation is typically carried out using a standard genetic code, which maps DNA nucleotides to the corresponding amino acids.

After translation, the HMM search is performed. This step involves using the HMM algorithm to compare the protein sequences obtained from the Multiple Sequence Alignment with the known protein sequences in the sequence database. The HMM algorithm uses probabilistic models to identify similarities and differences between the sequences being compared.

The search results are then filtered to remove false positives and irrelevant matches. This step is crucial for improving the accuracy of the analysis. The filtered results are then visualised to provide a clear and concise representation of the data.

Overall, the HMM analysis workflow proposed by Ustyantsev et al. [170] provides a reliable and efficient method for studying protein sequences and structures. The use of a RESTful API and docker images allow for easy deployment and access to

the HMM algorithm. Generally, an end-user can use the graphical user interface to add pipelines but can also use a PUT REST call which could be of the form

```
<backendAdress>/job
```

With the appropriate JSON body as in listing A.3.

Expererimental hidden Markov model search results

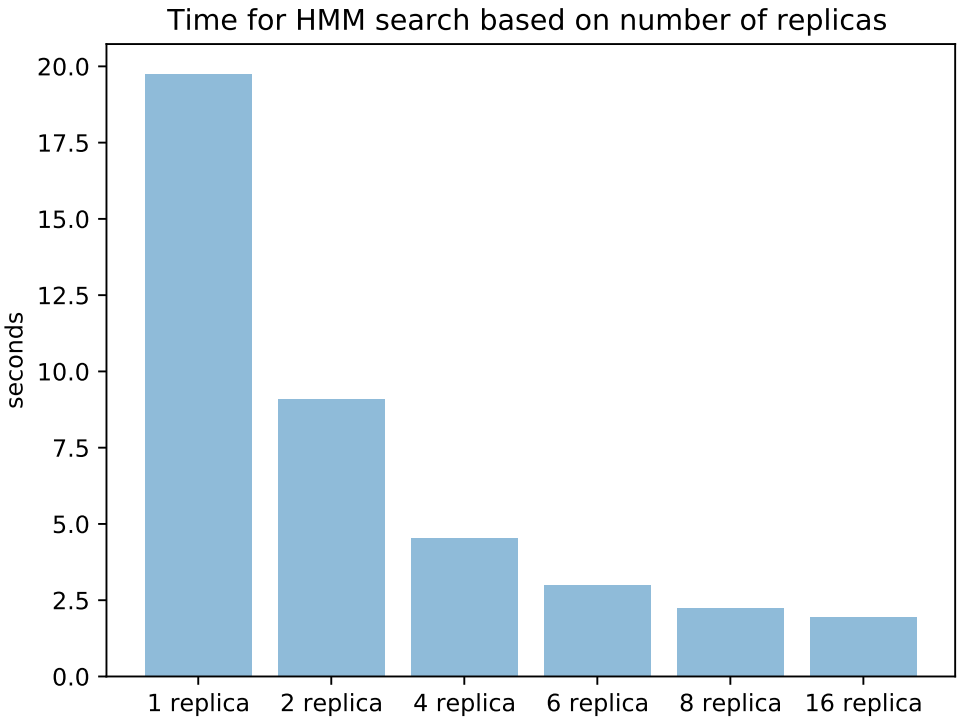


Figure 4.7: We ran a benchmark test for an HMM search as mentioned in the section Hidden Markov model search on prototypic sequences representing repetitive DNA from different eukaryotic species. The work is partitioned into parts and distributed to workers/replicas on a Docker Swarm cluster. Dependent on the number of replicas, the time to finish an HMM search reduces accordingly. Unparallelisable parts such as file loading and saving remain constant following Amdahl's Law [77] .

In addition to testing the ability to handle the increased workload for the process in a container, the benchmark looks at the possibility of splitting the workload

across multiple processors. Data parallelisation is a technique that allows for data distribution across multiple processors, which can increase the speed and efficiency of the computation. In the case of the HMM search, the data is split into roughly equal segments, and the search is performed using several replica sets. Each replica set is run on a separate processor, allowing for the parallelisation of the search.

The analysis is performed on a consumer's Personal Computer with an Intel-i7-5600 processor and 12 GB of memory. This processor is a quad-core model with a clock speed of 3.2 GHz, which is typical for a mid-range consumer PC. Each containerised HMM search is assigned 0.5 of a single core, which means that two HMM searches can be run simultaneously on each processor.

Figure 4.7 shows the time reduction when more replica sets are used on a partitioned dataset to perform the search. As the number of replica sets increases, the time for each job decreases. This result demonstrates the effectiveness of data parallelisation in speeding up the HMM search. However, a portion of the operation remains sequential, as proposed by Amdahl's Law [77]. This proportion mainly includes the data reading, calculating setup, and the Java wrapper, which cannot be parallelised.

After measuring the computation times for the sequential and parallelisable parts, it is found that a fraction $p = 0.93$ of the HMM search task can be parallelised. This result means that nearly all of the HMM search can be sped up through data parallelisation, with only a small portion remaining sequential. They are compared to the theoretical speedup values predicted by Amdahl's Law to validate these measured times. Amdahl's Law is defined as

$$S(r) = \frac{1}{(1-p) + \frac{p}{r}}$$

, where $S(r)$ is the theoretical speedup, p is the fraction that can be parallelized, and r is the number of processors/workers.

The measured results match the theoretical values, as shown in figure 4.8. In some instances, the measured results show slightly better speedup, possibly due to measuring inaccuracies in milliseconds. Overall, the benchmark results demonstrate the potential for data parallelisation to improve the performance of the HMM search algorithm.

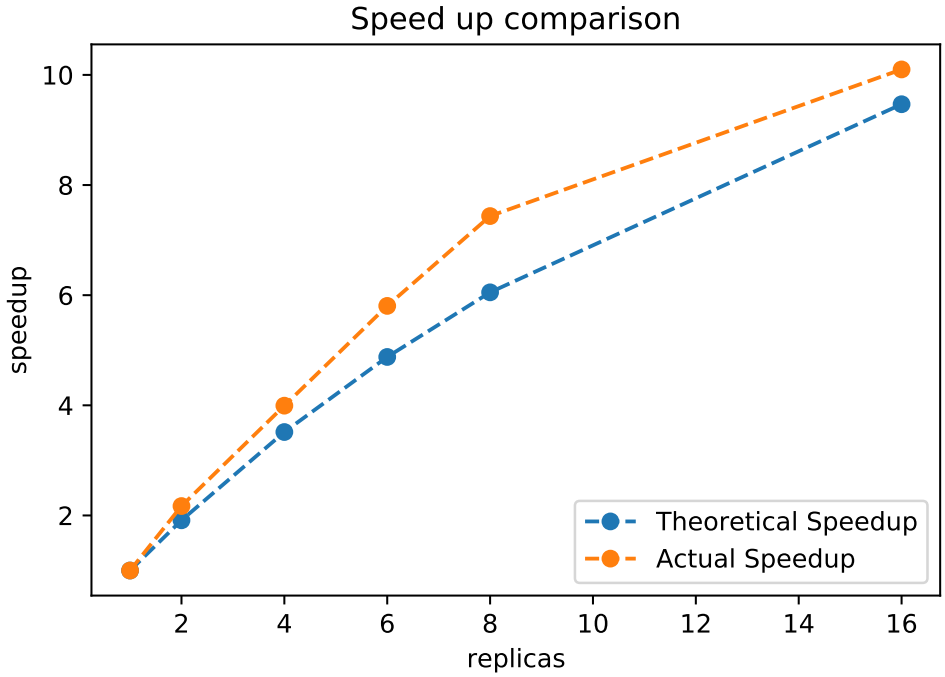


Figure 4.8: For the HMM search, we compared the speed up between our measured results and theoretical values. The theoretical values are calculated based on Amdahl's Law. We measured the serial and parral portion of the task. The task consists of a 0.93 parallelisable portion. Our results confirm the theoretical speedup for a task. The slight deviation might be due to measuring inaccuracies in milliseconds.

In conclusion, this study has proposed a Hidden Markov Model (HMM) analysis workflow for investigating protein sequences and structures. Furthermore, this study has proposed a hidden Markov modelanalysis workflow based on the methodology developed by Ustyantsev et al. [170]. This workflow, which utilises a RESTful API and docker images, provides a robust and efficient approach for investigating protein sequences and structures. Furthermore, the results of the benchmark analysis revealed the potential for data parallelisation to enhance the performance of the HMM search algorithm. A parallelisable fraction of 0.93 suggests that a significant proportion of the HMM search process can be accelerated through data parallelisation. Only a minimal portion remains sequential. These findings are in agreement with the predictions of Amdahl's Law, as the measured results match the theoretical values.

Overall, the proposed HMM analysis workflow is a valuable tool for researchers and scientists in bioinformatics and computational biology.

4.3.3 Time series analysis with convolutional long short-term memory neural networks

Biomedicine is a rapidly growing field that involves the study of the human body and its functions, as well as the development of medical technologies and treatments for diseases and disorders. As the field advances, it generates an increasingly large amount of data from various sources, including medical imaging, genetic sequencing, clinical trials, and electronic medical records. This data is essential for medical research, diagnosis, and treatment, but it can be challenging to manage and analyse due to its size, complexity, and heterogeneity.

One major issue in biomedicine is the need to organise and manage large amounts of data, including audio and video files, text-based databases, and proprietary medical file formats such as Digital Imaging and Communications in Medicine (DICOM) [131]. These data sources often have different structures and formats, making it challenging to integrate and analyse them together [124]. In addition, the data may be sensitive and confidential, requiring secure storage and access control [132].

our system addresses these challenges by using containerised environments to store and organise different types of data, including relational databases, object storage, and time-series databases [150]. This the approach allows data to be managed and accessed in a consistent and secure manner, providing a foundation for advanced data analysis and decision-making in biomedicine.

Specifically, time-series databases have the potential to provide new insights for diagnosis in biomedicine through the use of techniques such as knowledge discovery in databases [123]. For example, Anguera et al. Used data mining methods for epilepsy diagnosis in the electroencephalography domain [18]. Furthermore, Machine learning techniques have been used in biomedical research to analyse large and complex datasets [158].

In conclusion, biomedicine is a rapidly growing field that generates an increasing amount of data, which can be challenging to manage and analyse. Our system

uses containerised environments to store and organise different data types. This design decision allows for advanced data analysis and decision-making in biomedicine. Time-series databases and machine-learning techniques have the potential to provide new insights for diagnosis through the use of techniques such as knowledge discovery in databases and data mining methods. In this section, we aim to examine the integration of a new dataset in contrast to conducting a stress test on a convolutional neural network. Our objective is to demonstrate the capability of our system to integrate fully independent and deployable databases seamlessly.

Time series databases

Time series analysis is a powerful tool for addressing various challenges in biomedicine. Three of the main problems that time series analysis solves are

- Evaluating the effectiveness of treatments: By analysing time series data, healthcare professionals can determine whether a particular treatment is effective or not, and make adjustments as needed to improve patient outcomes.
- Making more informed decisions about a patient's care: Integrating data from multiple sources into a single time series database, healthcare professionals can gain a more a comprehensive view of a patient's health, allowing them to make more informed decisions about their care.
- Developing predictive models: By analysing time series data, researchers can develop predictive models that can be used to forecast a patient's future health, identify potential risk factors, and develop personalised treatment plans.

The critical aspect of time series analysis is its storage in a time series database. One of the main advantages of using a time series database in biomedicine is the ability to track changes in a patient's health over time [187]. This type of data can be used to monitor the effectiveness of treatments, identify trends or patterns in a patient's health, and identify potential problems or complications before they become serious. By analysing time series data, healthcare professionals can make more informed decisions about a patient's care and improve the overall quality of care.

Another benefit of time series databases in biomedicine is the ability to integrate data from multiple data generating sources [91]. This integration is particularly important in the healthcare industry, where data is often collected from a variety of sources, including medical records, wearable devices, and diagnostic tests. By integrating this data into a single time series database, healthcare professionals can gain a more comprehensive view of a patient's health and make more accurate predictions about their future health.

Overall, time series analysis is an essential tool for addressing a range of challenges in biomedicine. By enabling healthcare professionals to track changes in a patient's health over time, make more informed decisions about their care, and develop predictive models, time series analysis, has the potential to significantly improve the quality of care and patient outcomes in the healthcare industry.

Convolutional Long Short-Term Memory neural networks

Convolutional Long Short-Term Memory (ConvLSTM) is a type of deep a learning model that combines the features of convolutional neural networks (CNNs) and long short-term memory (LSTM) networks. It is commonly used for analysing time series data, especially in the field of biomedical engineering.

The convolutional part of the model allows it to automatically learn spatial hierarchies, which is vital for analysing images and other data with spatial structure. The LSTM part of the model allows it to maintain an internal state and learn from long sequences of data, making it well-suited for modelling time series data.

Convolutional Long Short-Term Memory (ConvLSTM) networks have been used in bioinformatics for various tasks; one example is the prediction of open chromatin regions from DNA sequences, where a recent study by Min et al. used this approach and integrated k-mer co-occurrence information with deep learning [121]. Another example is the prediction of protein secondary structure, wherein a a recent paper by Zhao et al., an optimised convolutional neural network and long short-term memory neural network models (OCLSTM) were applied [190]. The OCLSTM uses an the optimised convolutional neural network to extract local features between amino acid residues and a bidirectional long short-term memory neural network to extract remote interactions between the internal residues of the protein sequence to predict

the protein structure.

The theoretical background of ConvLSTM is the combination of features from convolutional neural networks (CNNs) and long short-term memory (LSTM) networks. Mathematically, these two model types can be described as follows:

A convolutional neural network applies a series of convolutional operations to an input matrix $X \in \mathbb{R}^{m \times n}$ to produce a new matrix $Y \in \mathbb{R}^{p \times q}$. The convolutional operation is defined as

$$Y_{i,j} = \sum_{k=1}^{K} \sum_{l=1}^{L} X_{i+k-1,j+l-1} \cdot W_{k,l}$$

where $W \in \mathbb{R}^{K \times L}$ is the kernel or filter matrix, and K and L are the height and width of the kernel, respectively.

A long short-term memory (LSTM) network maintains an internal state vector $h_t \in \mathbb{R}^n$ at each time step t, which is updated based on the current input $x_t \in \mathbb{R}^m$ and the previous state h_{t-1}. The update is controlled by a series of gates, including an input gate i_t, a forget gate f_t, and an output gate o_t. The new state h_t is calculated as

$$h_t = o_t \odot \tanh(c_t)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

where W_c, U_c, and b_c are trainable parameters, and $\odot$ denotes element-wise multiplication.

ConvLSTM combines these two model types by applying convolutional operations to the input data before feeding it into the LSTM network. This combination allows the model to learn spatial automatically hierarchies from the input data while still being able to maintain an internal state and learn from long sequences of data.

Experiments on diastolic blood pressure time series

In this study, we sought to investigate the use of a time-series forecast for the analysis of diastolic blood pressure. This method of analysis has potential applications in the

medical field, as it can provide valuable insights into a patient's health and assist in the early detection of potential health issues.

To perform this analysis, we employed a convolutional Long Short-Term Memory (LSTM) network, an approach first introduced by Shi et al. in 2015 [156]. LSTM networks have been shown to be effective at modelling time series data and are well-suited for tasks involving sequence analysis and prediction. In our adapted algorithm, we incorporated LSTM layers into a convolutional neural network architecture, which allows the model to extract local features from the input time series and integrate them into a global representation.

Our adapted algorithm was designed to accept a univariate time series as input, and we used patient data provided by Lin et al. in [110] for our analysis. The data consisted of a series of measurements of diastolic blood pressure taken over a period, providing us with a comprehensive dataset to train and evaluate our model.

Figure 4.9: The figure shows the integration of a time series database with a convolutional Long-Short-Term Memory network for time series forecasts. The blue boxes represent the different parts of the process, including the forecast visualisation, which is achieved by querying the original data from the database and plotting it together with the forecast. This separation allows for a more comprehensive analysis and a better understanding of the data. The resulting graph can be seen in Figure 4.10.

Figure 4.9 illustrates the data flow for our analysis task. The database and analysis steps in this figure are represented as green and blue boxes. These components are contained within containerised environments, which allows them to operate as loosely coupled components that can communicate via RESTful API. Furthermore, this approach allows us to integrate the analysis task into our existing software system without needing additional infrastructure or dependencies.

To ensure that our analysis is reliable and reproducible, we made use of con-

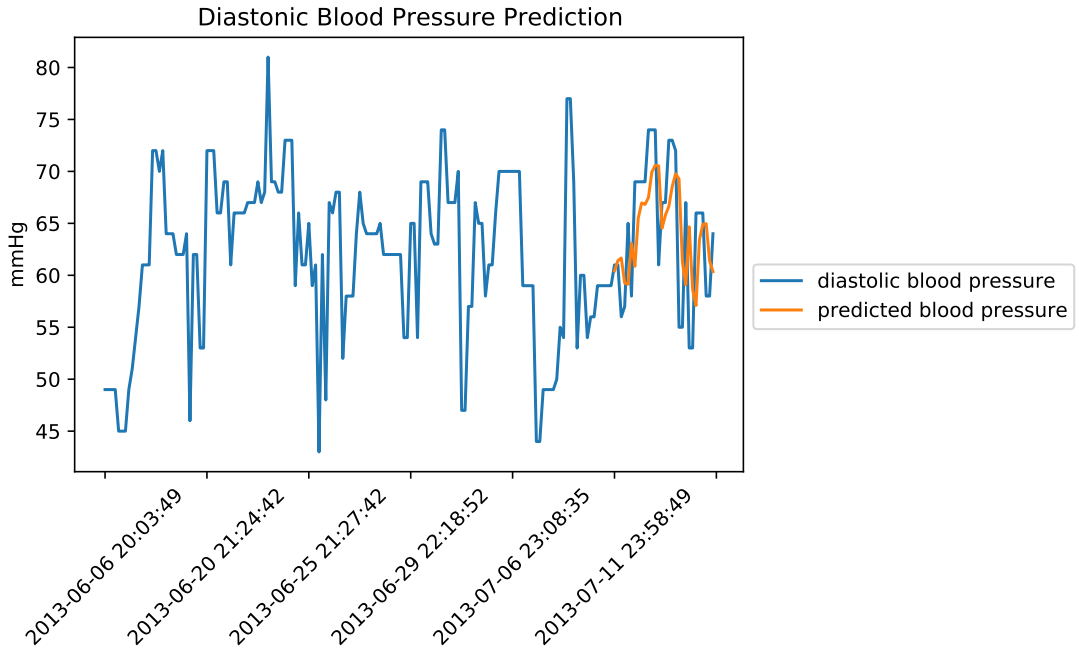


Figure 4.10: The figure shows a visualisation of the prediction made using an LSTM-CNN model on diastolic blood pressure data from a time-series database. The orange line represents the predicted values, while the blue line represents the actual diastolic blood pressure measurements. This prediction is discussed in more detail in the section titled Time series analysis with convolutional long short-term memory neural networks

tainerisation technology to package the necessary dependencies, libraries, and packages within the containerised environments. This encasement allows us to provide a consistent software environment for our analysis task, regardless of the underlying operating system or hardware.

For our time-series database, we used the OpenTSDB implementation [75]. This open-source time series database is designed for handling large volumes of data and provides efficient query capabilities, making it well-suited for our analysis task. The analysis steps were implemented as Python scripts contained within a Java wrapper, which handles RESTful queries like the HHM search described in Section Hidden Markov model search on prototypic sequences representing repetitive DNA from different eukaryotic species. The resulting prediction is plotted in Figure 4.10.

The integration of this time-series database demonstrates our system's flexibility.

As data mining becomes increasingly important, it is essential for software systems to be able to accommodate a wide range of data types. With the continual improvement of data generation methods

The integration of this time-series database demonstrates our system's flexibility. As data mining becomes increasingly important, it is essential for software systems to be able to accommodate a wide range of data types. With the continual improvement of data generation methods, medical diagnoses may rely on previously inaccessible data. Furthermore, biomedical scientists require reliable and reproducible software environments. Unfortunately, installing all necessary tools and databases on the operating system can cause conflicts. However, our loosely coupled systems provide a solution by allowing for storing a wide range of data in separate environments without interference.

4.4 Implications, Challenges, Limitations and Future Research Directions

Following our experiments, we will proceed to the discussion section. In this section, we will analyse the interpretation and context of the experiments, as well as any limitations encountered. Additionally, we will evaluate the broader implications of our findings and compare and contrast our experiments with other bioinformatics systems.

4.4.1 Interpretation and context

This chapter focussed on three main research questions.

- How does the implementation of parallel computation techniques in our framework improve the speed of data analysis in bioinformatics?
- What are the most effective approaches for integrating and analysing data from multiple sources within our framework in bioinformatics?
- How can our framework be scaled to handle effectively large-scale datasets in bioinformatics, and how can improve machine learning techniques are applied

to optimise data integration and analysis in this context?

In our experiments, we explored the use of various bioinformatics methods that are commonly applied in their respective fields, including unsupervised matrix factorisation, predictive convolutional long short-term memory (LSTM) neural networks and hidden Markov model (HMM) search to optimise data integration and analysis in our framework. Our findings reveal that our system can significantly reduce computational time through the use of both task and data parallelisation, which allows for faster processing of data, which is essential in the field of bioinformatics, where large datasets are often encountered. In addition, our system is also able to integrate multiple types of databases within a closed environment, allowing for seamless integration and management of different types of data, which is essential for conducting comprehensive analyses. Furthermore, We also implemented a docker swarm environment to scale our framework to handle large-scale datasets by distributing the workload across multiple machines, we could process large datasets more efficiently and effectively. The machine learning techniques (unsupervised matrix factorisation, predictive convolutional LSTM neural networks, and HMM search) we used to identify patterns in the data and make predictions about future data. This result allowed us to analyse and interpret the data, which is critical for conducting comprehensive analyses in bioinformatics. Overall, our findings highlight the the usefulness of our system in addressing challenges, such as the implementation of parallel computation techniques in bioinformatics.

Our system addresses the second research question, which asks about approaches for integrating and analysing data from multiple sources within our framework in bioinformatics by using a docker environment with a RESTful API to integrate data from various sources. The docker environment provides a consistent and standardised environment for running our system, making integrating data from different sources easier. Furthermore, the RESTful API allows our system to communicate with other applications and systems over the network, enabling it to access and integrate data from multiple sources. Our system uses these approaches by seamlessly integrating and analysing data from multiple sources, enabling comprehensive analyses. We have demonstrated this approach's effectiveness through experiments using various data sources, including time series data, multiple sequence alignment data, and gene

expression data. Additionally, we have reviewed the literature on data integration and analysis approaches in bioinformatics and evaluated the effectiveness of these approaches within our system. Our findings highlight the usefulness of our system in addressing challenges in the field of bioinformatics related to integrating and analysing data from multiple sources.

In addition to improving computational efficiency, our system can also integrate multiple types of databases within a closed environment. This feature is valuable as it allows for the seamless integration and management of different data types, which is essential for conducting comprehensive analyses. Data integration is a crucial challenge in bioinformatics, as it combines data from multiple sources and formats to enable meaningful analysis. Our system's ability to effectively integrate data from multiple sources may enable researchers to access and use diverse data sets more efficiently, leading to more comprehensive and accurate analyses.

Furthermore, our system can effectively handle large datasets using various methods. This management is critical as it allows for the accurate and efficient analysis of large amounts of data, often required in bioinformatics research. Large datasets are standard in bioinformatics, and analysing them can be complex and time-consuming. Our system's ability to effectively handle large datasets may enable researchers to more quickly and efficiently analyse these datasets, which may facilitate more comprehensive and accurate research.

Overall, our findings highlight the usefulness of our system in addressing challenges in the field of bioinformatics. Its ability to improve computational efficiency, integrate multiple types of databases, and effectively handle large datasets, making it a valuable tool for researchers working in this field. By enabling faster and more efficient data analysis, our system may facilitate the advancement of bioinformatics research and the development of new insights and technologies in this field.

4.4.2 Limitations and bias

Our research using matrix decomposition on GTEx datasets, HMM search on Repbase, and CNN on time series has limitations. However, these experiments are sufficient for answering our research questions. We have addressed the limitations in a way that allows the experiments to provide sufficient answers. Despite the

limitations, we are confident in the value and usefulness of our research.

One limitation of this study is the sample size. We used a subset of plants and fungi from the Repbase database and brain tissues from the GTEx datasets. While this sample size may be sufficient to answer our research questions, expanding the data to include more data types (e.g. vertebrates, primates, or tissues) or a more extensive data sample could affect the results and performance. A small sample size may not be representative of the larger population and may not have enough statistical power to detect significant differences. However, our experiments were designed to address specific research questions, such as the software architecture components for bioinformatics data analyses for small to medium research teams and may not require a large sample size to do so, considering the specific research question and available resources.

Another limitation of this study is the quality of the data. Imprecise data can affect the accuracy and reliability of the results, leading to potentially misleading or incorrect conclusions. To address this concern, it is crucial to carefully select and evaluate the data to ensure that it is of high quality and appropriate for the research objectives. In our research, we paid particular attention to the selection and evaluation of the data. Our approach minimises the impact of this limitation on our experiments. We used the GTEx and Repbase datasets, widely recognised as reliable genomic and reference sequence data sources. We also applied quality control measures — such as data preprocessing — to the data to ensure its accuracy and reliability. Despite these efforts, some inaccuracies or biases may still be present in the data, which could affect the results of our experiments. However, the data we used is of sufficient quality to provide valuable insights into our research questions.

Additionally, it is crucial to accurately consider and control for confounding variables to identify the relationship between the independent and dependent variables. In this study, we identified two potentially confounding variables: the algorithms and techniques used and the experience and expertise of the study's researchers. Regarding the algorithms and techniques used, we chose to utilise HMM search and several matrix decomposition methods, which are classic techniques for bioinformatics data analysis. However, there are several other methods that could potentially be considered, such as multidimensional scaling and randomised sampling [151, 169].

To control for this confounding variable, we carefully evaluated the suitability of the chosen algorithms and techniques based on the specific research objectives and available resources. In terms of the experience and expertise of the researchers, the field of bioinformatics is vast and encompasses topics in software engineering, biology, and chemistry. While we sought to gain a deep understanding of these fields, a more profound understanding may have affected the results of the experiments. Therefore, to control for this confounding variable, we carefully selected our research team to ensure that we had expertise in all relevant fields. In addition, we conducted a thorough literature review to ensure that we were aware of the latest developments in the field. By carefully considering and controlling for these confounding variables, we were able to produce reliable and accurate results that accurately reflect the relationship between the independent and dependent variables in this study.

Our research may also be limited by the need for more consideration of algorithmic parallelisation to improve scalability. Algorithmic parallelisation involves designing algorithms that can be parallelised. This design allows the algorithms to be processed simultaneously by multiple computing resources [152]. This partition can be an effective strategy for improving the efficiency and speed of a system, mainly when dealing with large datasets or complex tasks. However, it may involve using consensus methods, which can be complex and time-consuming. In our study, we did not explore algorithmic parallelisation or consensus methods, which may limit the scalability of our approach. While algorithmic parallelisation is an essential technique for improving scalability, it may only sometimes be practical or necessary, depending on the specific research objectives and available resources.

Overall, while our research is not without limitations, we believe that the experiments we conducted were well-suited to address our research questions and provide valuable insights. By carefully considering the limitations and biases of our approach and taking steps to address them appropriately, we were able to produce high-quality research that is useful and relevant to our field.

4.4.3 Compare and Contrast

Other bioinformatics systems focus on our research question. These questions revolve around three main aspects: Parallelisation, data integration and scalability. We

highlight several approaches to handle these challenges by comparing and contrasting our proposed system with them.

One common goal shared by several bioinformatics tools is the efficient management and analysis of large amounts of data. Some of the tools are the parallel package in `R`, Atlas and BioWarehouse, and Pachyderm. The `R/parallel` package allows users to perform parallel computations in `R`, which can help speed up analyses on extensive data sets [58,175]. Atlas and BioWarehouse provides centralised repositories for storing and managing biological data, which can be accessed and analysed by researchers [105,154]. Pachyderm helps manage the flow of data between different processes and systems, which can be helpful for data science and machine learning workflows that involve large amounts of data [129].

There are also several challenges that these tools can help address. One challenge is handling large data sets, which can be time-consuming and resource-intensive. The parallel package and Pachyderm can help address this challenge by providing efficient methods for processing and managing data. Another challenge is ensuring the reproducibility of data analyses, which is vital for maintaining the integrity of research results. Pachyderm can help with this by providing a way to track data history and ensure that data pipelines are reproducible. Additionally, managing the flow of data between different processes and systems can be challenging, especially when working with large amounts of data. Pachyderm can help with this by providing a way to orchestrate data flow and ensure that data is moved between processes and systems in a controlled and reproducible manner.

Our proposed bioinformatics system shares the goals above, including efficiently managing and analysing large amounts of data. Additionally, our system aims to address the challenges of handling large data sets, ensuring the reproducibility of data analyses, and managing the data flow between different processes and systems. Our system achieves these goals by utilising advanced algorithms and data management techniques to effectively process and analyse large amounts of data. Furthermore, we have implemented measures to ensure the reproducibility of data analyses, such as tracking the data history and controlling the data flow between different processes and systems. By addressing these challenges, our system aims to provide a robust and reliable platform for conducting bioinformatics research.

Functionality

In data analysis, several tools and resources are available to facilitate efficient and effective workflows. One such tool is the Parallel package in`R`, which allows users to perform parallel computations in`R`, significantly speeding up analyses on large data sets. It can handle various data types, including numeric, categorical, and text, and can be used for statistical modelling, machine learning, and data visualisation.

Centralised repositories, such as Atlas and BioWarehouse, help store and manage biological data. These resources can handle various data types, including genomic, proteomic, and transcriptomic data, and can be utilised for gene expression analysis, pathway analysis, and variant annotation.

Pachyderm is a tool that streamlines the flow of data between different processes and systems, particularly in the context of data science and machine learning workflows involving large amounts of data. It is capable of handling structured and unstructured data and can be used for tasks such as data preparation, transformation, and model training and evaluation.

In summary, these tools and resources provide valuable support for data analysis, enabling parallel computations, efficient data management, and streamlined data flow. They are instrumental in facilitating research and advancing the field of data science.

Our proposed system leverages the power of parallelisation, as exemplified by the Parallel package in`R`, to significantly reduce the time required for data analysis, mainly when working with large datasets. By utilising multiple processing units simultaneously, we can achieve a much faster turnaround time for statistical modelling, machine learning, and data visualisation tasks.

In addition to the benefits of parallelisation, our system also incorporates an object storage system similar to Pachyderm, which allows for efficient management of data flow between different processes and systems. This design is particularly useful in the context of data science and machine learning workflows, where large amounts of data are often involved. The object storage the system allows for the seamless integration of structured and unstructured data, enabling a wide range of analyses, including data preparation, transformation, and model training and evaluation.

Overall, the parallelisation and robust data management system make our pro-

posed system a powerful tool for efficient and effective data analysis. As a result, it can significantly accelerate research and advance data science.

Performance

In a study by Vera et al., the use of the `R/parallel` package was found to improve the performance of a multiple QTL model significantly approach for analysing gene expression data [175]. Specifically, the authors reported that the execution time of their analysis was reduced from approximately 3 hours to 1 hour by using four workers. This result is similar to the performance improvements observed in our proposed system when applying the HMM search with multiple workers. These findings suggest that parallelisation techniques can effectively optimise the performance of computational analyses in bioinformatics, particularly when analysing large datasets.

In a study by Novella et al., the authors conducted a performance test on a larger scale using OpenNMS's FeatureFinderMetabo and find peaks tools on 137 samples [129]. They found that the speedup for a cluster of 19 workers was approximately 17, as calculated using the following formula:

$$S(N) = \frac{T_0}{T_N}$$

where T_0 is the serial running time and T_N is the parallel running time using N processing elements.

In comparison, our experiments had a speedup of approximately 8, calculated using the same formula. Several factors may contribute to the difference in the results between the two studies. For example, the hardware architecture used in the study by Novella et al. (a node with seven vCPUs and 32 GB of RAM) may have differed from the hardware used in our experiments. Additionally, the specific software implementation of the tools in question may have affected the speedup achieved.

Despite these differences, it is essential to note that both studies were able to demonstrate the effectiveness of using multiple processing elements to improve the performance of these bioinformatics tools. Although achieved on a smaller scale, our results are still valuable and worthy of consideration.

Data and tool integration

Bioinformatics frameworks can vary in terms of their focus and expandability when it comes to tool and data integration. The`R` programming language's parallel package is a tool designed for parallel computing within an `R` environment, capable of handling various data formats native to`R`, including CSV, TSV, and Excel files, as well as specialised formats like FASTA and FASTQ for biological sequence data. While the parallel package may be used in conjunction with other`R` packages or tools, it may not be easily integrated with systems or tools using different technologies or programming languages.

Atlas and BioWarehouse are centralised data repositories that store and manage biological data. They support a range of formats commonly used in the life sciences, such as FASTA, FASTQ, BAM, and VCF, as well as proprietary formats specific to particular research groups or data sources. In addition, these repositories offer interfaces for accessing the data from external systems, making them useful for integrating with other tools or systems that need to access or analyse the data.

Pachyderm is a tool for managing data pipelines and workflows. It uses a declarative model to specify the input data, processing steps, and output data for each pipeline stage. The declarative model is a way of defining the desired state of a system, in this case, the input, processing, and output of a data pipeline, rather than specifying how to achieve that state. This implementation allows Pachyderm to handle a variety of structured and unstructured data types and to manage and process data across multiple stages of a pipeline, making it a valuable tool for cohesively integrating data and tools. Pachyderm may also be integrated with other systems or tools using APIs or other interfaces, or by using data storage systems like object stores or databases that can be accessed by multiple systems.

Our system is designed to facilitate the integration of various data types and tools to support data analysis and processing pipelines. Like Pachyderm, our system utilises a docker architecture to achieve this integration. This choice allows us to include a wide range of data and methods in our pipelines. Furthermore, it ensures that these elements are portable and easy to deploy in various environments.

The use of a docker architecture in our system also allows us to quickly expand the range of tools and data types that can be incorporated into our pipelines. This

expansion is possible because docker containers can be used to package and deploy any tool or data type that can be run in a containerised environment, giving us the ability to include a wide range of methods and data types in our pipelines.

Overall, the use of a docker architecture in our system makes it particularly well-suited for integrating a diverse range of data types and tools, and allows us to offer greater flexibility and adaptability in the types of analyses and processing pipelines that can be supported. This design makes our system a valuable asset for researchers and analysts are seeking to manage and analyse complex data sets effectively.

Ease of use

Software setup frameworks often vary in their installation steps and requirements. The parallel package is one example of a software package that can be easily installed and used within anR environment. This package allows for parallel processing within theR programming language and requiresR to be installed, as well as knowledge of theR language's syntax.

In contrast, setting up centralised repositories such as Atlas and BioWarehouse involves a more complex process. This process may include installing and configuring server software, establishing database systems, and implementing security protocols. Additionally, users must be familiar with the data formats and protocols used by these repositories to store and access data. The underlying architecture of these tools requires the establishment and maintenance of infrastructure, which can be a more involved process than the setup of some other software packages.

Pachyderm is a tool that can be used to manage data pipelines and workflows. It utilises the Kubernetes container orchestration system to execute data processing tasks. Setting up Pachyderm involves installing and configuring the Pachyderm software and establishing a cluster of machines to run it on. Familiarity with the Pachyderm data model and the command-line interface is also necessary to use this tool. Once Pachyderm is installed on a Kubernetes cluster, users can create and run data pipelines using the Pachyderm command-line interface or APIs. Pachyderm pipelines are defined using a declarative model that outlines each pipeline stage's input, processing steps, and output data. Pachyderm utilises Kubernetes executes the pipeline stages as containerised tasks, which can be run in parallel or sequentially

as needed. The underlying architecture of Pachyderm involves the use of container orchestration, which enables the parallel or sequential execution of pipeline stages.

Our proposed system utilises a Docker Swarm architecture to deploy various services. This architecture offers several benefits compared to other options, including ease of setup and low resource requirements. To set up the Docker Swarm architecture, users must install Docker on each machine in the cluster. This process is relatively straightforward and can be accomplished by using a package manager or downloading the Docker installation package from the official website.

Once Docker is installed on all of the machines in the cluster, users can create and run data pipelines using the Docker Swarm command-line interface or APIs. These pipelines are defined using a declarative model that outlines each pipeline stage's input, processing steps, and output data . Docker Swarm utilises containerisation to execute the pipeline stages, allowing for the efficient execution of tasks in parallel or sequentially as needed.

One advantage of Docker Swarm for data pipeline management is its low resource requirements. The Docker Swarm architecture utilises a lightweight containerisation approach, which allows for efficient resource utilisation and improved performance compared to traditional virtualisation techniques. Additionally, containerisation allows for easier scaling and deployment of services, as containers can be easily moved between machines or clusters.

In summary, the use of a Docker Swarm architecture for the data pipeline management offers several benefits, including ease of setup, low resource requirements, and improved performance. Users can leverage the Docker Swarm command-line interface or APIs to create and run data pipelines, utilising a declarative model to specify the input data, processing steps, and output data for each stage of the pipeline. Overall, the Docker Swarm architecture offers a powerful and flexible solution for managing data pipelines and workflows.

4.5 Conclusion

In our experiments, we examined the use of various bioinformatics methods to address challenges in the field. Our system can significantly reduce computational

time through task and data parallelisation, making it more efficient for processing large datasets encountered in bioinformatics. In addition, our system can seamlessly integrate and analyse data from multiple sources using a docker environment with a representational state transfer based application programming interface, enabling comprehensive analyses.

We also applied machine learning techniques such as unsupervised matrix factorisation, predictive convolutional long short-term memoryneural networks, and hidden Markov modelsearch to optimise data integration and analysis in our framework. These approaches have shown promise for identifying patterns in data and making predictions, but their effectiveness may be limited by the quality and nature of the data being analysed. Additional experiments will expand the usefulness of our proposed system. To scale our framework to handle large-scale datasets, we relied on our implemented system with a docker swarm environment to distribute the workload across multiple machines as described in chapter *3 Advanced Software Engineering in Bioinformatic: A Case Study of Design and Implementation*.

Overall, our findings demonstrate the usefulness of our system for optimising data integration and analysis in bioinformatics. Its ability to improve computational efficiency, integrate multiple types of databases and effectively handle large datasets, making it a a valuable tool for researchers in this field. Further research may be necessary to evaluate the effectiveness of these approaches in different contexts and to identify additional methods for optimising data integration and analysis.

Chapter 5

Discussion: A comparative analysis of summary, limitations, and comparison

In this chapter, we will summarise the results of our study on the impact of software patterns, microservices, Docker containers, and advanced software engineering principles and practices on bioinformatics software systems. We will also discuss the limitations and biases of our research and provide context by comparing our work to related studies in bioinformatics.

5.1 Introduction

Developing bioinformatics software systems is a complex task that requires the integration of various technologies and approaches. In this study, we aimed to investigate the potential benefits of implementing software patterns, microservices, Docker containers, and advanced software engineering principles and practices in developing such systems. Through a series of experiments, we implemented and evaluated a system that utilised a docker environment with a RESTful API to integrate data from various sources and apply bioinformatics methods. Our findings suggest that using these technologies can significantly enhance bioinformatics software systems' reliability, scalability, performance, efficiency, and productivity. However, it is vital to consider the study's limitations, including the limited scope, the vast amount of tools and APIs to consider, time and resource constraints, the potentially small sample size, the possibility of biases or inaccuracies in the data, and potential confounding variables. As a result, the study's results may not generalise to other data types or a larger population. In addition, there may be bias in the selection and evaluation of the data and in the choice of algorithms and techniques. In this discussion chapter, we will summarise our results and examine the limitations and context of our research within bioinformatics.

5.2 Summary of results

The primary goal of this study was to explore the potential benefits of implementing various software patterns, microservices, and Docker containers, as well as advanced software engineering principles and practices in the development of bioinformatics software systems. Our research aimed to assess the impact of these technologies on

the reliability, scalability, performance, efficiency, and productivity of bioinformatics software systems.

In order to address this research question, we conducted a series of experiments and analysed the results to identify trends and patterns. Our findings suggest that using software patterns such as the saga pattern and the Command Query Responsibility Segregation (CQRS) pattern can significantly enhance the reliability and scalability of bioinformatics software systems, respectively. In addition, microservices and Docker containers can improve system reliability, scalability, and performance.

Additionally, our research indicates that the adoption of advanced software engineering principles and practices, such as model-driven design and Docker orchestration with Swarm can facilitate the efficient and productive deployment and management of bioinformatics software systems. Furthermore, the implementation of a distributed architecture can enhance the efficiency and productivity of a small research group through the parallelisation of tasks and the the utilisation of additional resources.

Our experimental results showed that our system was able to significantly reduce computational time through the use of both task and data parallelisation. This reduction is a significant advantage as it allows for faster processing of data, which is essential in bioinformatics, where large datasets are often encountered. In addition to improving computational efficiency, our system was also able to integrate multiple types of databases within a closed environment effectively. This encapsulation is a valuable feature as it allows for the seamless integration and management of different data types, which is essential for conducting comprehensive analyses. Furthermore, our system was able to handle large datasets using a variety of methods. This integration is critical as it allows for the accurate and efficient analysis of large amounts of data, often required in bioinformatics research. Overall, our experimental results highlight the usefulness of our system in addressing challenges in the field of bioinformatics. Its ability to improve computational efficiency, integrate multiple types of databases and effectively handle large datasets make it a valuable tool for researchers working in this field.

In conclusion, our findings suggest that the incorporation of software patterns, microservices, Docker containers, and advanced software engineering principles and

practices can significantly enhance the reliability, scalability, performance, efficiency, and productivity of bioinformatics software systems. It is essential to consider the specific needs and constraints of the system, as well as the availability of resources and expertise when implementing these technologies in order to optimise the success of the system. Our system, which utilises a docker environment with a RESTful API to integrate data from various sources and apply bioinformatics methods, has demonstrated the ability to reduce computationally significantly time through task and data parallelisation integrate multiple types of databases within a closed environment and effectively handle large datasets. These capabilities make it a valuable tool for researchers in the field of bioinformatics.

5.3 Limitations and bias

This thesis has several limitations that should be considered when interpreting the results. One limitation was the limited scope of the research, which focused on a specific aspect of bioinformatics microservices and needed to fully explore other essential topics such as software engineering principles and data handling and tool integration for various analysis methods. These topics are essential for building a flexible data analysis system and are addressed in several research papers on bioinformatics software engineering. However, due to time and resource constraints, we could not investigate these topics in depth thoroughly.

Another limitation was the vast amount of tools and APIs that needed to be considered in order to make the system more versatile. There are several tools that incorporate microservice architecture, such as Refget, Gen3 Framework Service and the PhenoMeNal project, each with a different focus and approach to facilitating the sharing of clinical and genomic data and improving interoperability in the field of bioinformatics. Building a software architecture that can handle and interact with these tools is a challenge that must be carefully weighed and considered when designing the framework.

In addition to these limitations, the study's sample size may also be a concern. The study used a subset of data, which may not represent the larger population and may need more statistical power to detect significant differences. Therefore, it is

crucial to consider the potential impact of sample size on the reliability and validity of the results.

Another limitation to consider is the data quality used in the study. Some inaccuracies or biases may still be present in the data, which could affect the results of the experiments. To minimise the impact of this limitation, it is crucial to carefully select and evaluate the data to ensure that it is of high quality and appropriate for the research objectives. In our research, we paid particular attention to the selection and evaluation of the data and applied quality control measures to ensure its accuracy and reliability. However, it is still possible that some inaccuracies or biases may remain in the data.

It is also essential to consider and control for confounding variables to accurately identify the relationship between the independent and dependent variables. In this study, we identified and controlled for two potentially confounding variables: the algorithms and techniques being used and the experience and expertise of the researchers. However, there may be other confounding variables, such as environmental factors, that were not considered in the study. Therefore, it is essential to carefully consider and control these variables to produce reliable and accurate results.

Finally, the generalizability of the results should also be considered. The results of the study may not be applicable or relevant to other data types or a larger population. It is crucial to consider the potential impact of sample size and the characteristics of the sample on the generalizability of the results.

Despite these limitations, this thesis was able to successfully address them and produce valuable insights into the use of software patterns, microservices, Docker and advanced software engineering principles and practices in bioinformatics software systems. The research demonstrates the feasibility and effectiveness of these technologies in addressing the complexities and demands of bioinformatics software systems. Additionally, we implemented many concepts and principles that we studied, further supporting their usefulness in this context. While it is essential to consider the study's limitations, our research showcases the potential of these technologies in addressing the challenges of building reliable and scalable bioinformatics software systems.

5.4 Comparison to other works

Our work on using software patterns, microservices, Docker, and advanced software engineering principles and practices in bioinformatics software systems fits into the larger context of research on bioinformatics software engineering. Bioinformatics software systems are complex and require the use of advanced software engineering techniques in order to be reliable and scalable. Our research is guided by the goal of identifying and evaluating these techniques to inform the design and development of bioinformatics software systems.

This topic has been addressed in several research papers [21, 22, 83, 100], which have identified various challenges and opportunities in the field of bioinformatics software engineering. For instance. Lower highlights the lack of software engineering skills in the field of bioinformatics and proposes the creation of a new role called "Bioinformatic Engineer" as a solution. Lower also recommends cross-training software engineers in the life sciences and researching Domain Specific Languages to facilitate collaboration between engineers and bioinformaticians. [100]. As bioinformatics data analysis can be seen as a subset of data-centric analysis, similar challenges also apply. Hummel et al. emphasise the need for efficient algorithms and data structures in the development of data-centric systems in order to handle large amounts of data without experiencing performance degradation. They also note the importance of scalability, data storage, data processing, data analysis, and data visualisation in the field of data-centric analysis [83]. Bare et al. emphasise the use of software to facilitate the analysis of high-throughput data in the field of biology and the importance of interoperability between software tools. They propose using simple data structures such as lists, matrices, networks, tables, and tuples to achieve interoperability and provide guidelines for future software development. These challenges and considerations can impact the design of software architecture [21].

Our work builds upon and extends previous research in this area by providing a detailed analysis of the potential benefits and challenges of using microservices in bioinformatics software systems. Microservices are a software architecture pattern that involves the decomposition of a monolithic application into a set of small, independent services that can be developed, deployed, and scaled separately. We

evaluate the use of microservices in the context of bioinformatics software systems and identify several benefits, such as improved modularity, scalability, and flexibility. We also discuss the challenges of using microservices in this context, such as the need for effective service design and integration, and the potential overhead of managing a large number of services.

In addition to our theoretical analysis, we also present a case study in which we implement many of the concepts and principles that we studied. This case the study demonstrates the feasibility and effectiveness of using microservices and other advanced software engineering techniques in building bioinformatics software systems. Our research contributes to the growing body of knowledge on bioinformatics software engineering and highlights the potential of microservices and other advanced software engineering techniques in building reliable and scalable software systems for bioinformatics data analysis.

5.5 Conclusion

In conclusion, our study aimed to explore the potential benefits of implementing various software patterns, microservices, and Docker containers, as well as advanced software engineering principles and practices in developing bioinformatics software systems. Our findings suggest that these technologies can significantly enhance bioinformatics software systems' reliability, scalability, performance, efficiency, and productivity. Through a series of experiments, we implemented and evaluated a system that utilises a docker environment with a RESTful API to integrate data from various sources and apply bioinformatics methods. Our results demonstrated that the system could significantly reduce computational time through task and data parallelisation, integrate multiple types of databases within a closed environment, and effectively handle large datasets. These capabilities make it a valuable tool for researchers in the field of bioinformatics. However, it is vital to consider the study's limitations, including the limited scope, the vast amount of tools and APIs to consider, time and resource constraints, the potentially small sample size, the possibility of biases or inaccuracies in the data, and potential confounding variables. The study's results may not be generalisable to other data types or a larger population. There

may be bias in the selection and evaluation of the data, as well as in the choice of algorithms and techniques. Despite these limitations, our research contributes valuable insights into the use of software patterns, microservices, Docker and advanced software engineering principles and practices in bioinformatics software systems.

Chapter 6

Conclusion and outlook

In this chapter, we summarised the main findings of our study, which pertain to the impact of software patterns, microservices, Docker, and advanced software engineering principles and practices on the reliability, scalability, performance, efficiency, and productivity of bioinformatics software systems. We also discussed the implications and applications of these findings for researchers and practitioners in the field. We proposed several directions for future work, including the expansion of the system's REST API and the testing of different data sets and algorithms. Finally, we concluded the study by highlighting the key takeaways from our research and the value of our work in advancing our understanding of bioinformatics software systems.

6.1 Main findings

In this study, we investigated the potential benefits of using various software patterns, microservices, and Docker containers, as well as advanced software engineering principles and practices in developing bioinformatics software systems. Through a series of experiments, we found that implementing software patterns such as the saga pattern and the CQRS pattern can significantly improve the reliability and scalability of bioinformatics software systems, respectively. Microservices and Docker containers also enhanced system reliability, scalability, and performance. Our research also indicates that adopting advanced software engineerings practices, such as model-driven design and Docker orchestration with Swarm, can facilitate the efficient and productive deployment and management of bioinformatics software systems. Finally, we found that Our implementation of a distributed architecture improves the efficiency and productivity of a small research group through the parallelisation of tasks and the utilisation of additional resources.

Our experimental results showed that our system could significantly reduce computational time using task and data parallelisation. It was also able to effectively integrate multiple types of databases within a closed environment, allowing for the seamless integration and management of different types of data. Additionally, our system handled large datasets using various methods, which is critical for the accurate and efficient analysis of large amounts of data often encountered in bioinformat-

ics research. Overall, our system demonstrated the ability to address challenges in bioinformatics and is a valuable tool for researchers in this field.

6.2 Implications and application

The primary goal of this study was to explore the potential benefits of implementing various software patterns, microservices, and Docker containers, as well as advanced software engineering principles and practices, in developing bioinformatics software systems. In addition, our research aimed to assess the impact of these technologies on the reliability, scalability, performance, efficiency, and productivity of bioinformatics software systems. In order to address this research question, we conducted a series of experiments and analysed the results to identify trends and patterns.

Our findings suggest that using software patterns such as the saga pattern and the Command Query Responsibility Segregation (CQRS) pattern can significantly enhance the reliability and scalability of bioinformatics software systems, respectively. Microservices and Docker containers can improve system reliability, scalability, and performance. The use of microservices enhances the modularity and maintainability of a system. This architecture can lead to reduced downtime and increased stability. The application of Docker containers for the packaging and deployment of microservices facilitates the scaling and deployment of the system, contributing to improved scalability. Additionally, the adoption of object storage for data storage and REST for component communication can further enhance the performance and reliability of the system.

Furthermore, our research indicates that the adoption of advanced software engineering principles and practices, such as model-driven design and Docker orchestration with Swarm can facilitate the efficient and productive deployment and management of bioinformatics software systems. The integration of these practices can support the efficient and productive deployment and management of a large number of Docker containers. In addition, the implementation of a distributed architecture can enhance the efficiency and productivity of a small research group through the parallelisation of tasks and the utilisation of additional resources.

Our experimental results showed that our system could significantly reduce com-

putational time using task and data parallelisation. This parallelisation is a significant advantage as it allows for faster processing of data, which is essential in bioinformatics. Furthermore, researchers often encounter large datasets for their experiments. In addition to improving computational efficiency, our system was also able to integrate multiple types of databases within a closed environment effectively. This integration is a valuable feature as it allows for the seamless integration and management of different data types, which is essential for conducting comprehensive analyses. Furthermore, our system was able to handle large datasets using a variety of methods. This handling is critical as it allows for the accurate and efficient analysis of large amounts of data, often required in bioinformatics research. Overall, our experimental results highlight the usefulness of our system in addressing challenges in the field of bioinformatics. Its ability to improve computational efficiency, integrate multiple types of databases and effectively handle large datasets make it a valuable tool for researchers working in this field.

In conclusion, our findings suggest that the incorporation of software patterns, microservices, Docker containers, and advanced software engineering principles and practices can significantly enhance the reliability, scalability, performance, efficiency, and productivity of bioinformatics software systems. It is vital to consider the specific needs and constraints of the system, as well as the availability of resources and expertise when implementing these technologies in order to optimise the success of the system. Our system, which utilises a docker environment with a RESTful API to integrate data from various sources and apply bioinformatics methods, has demonstrated the ability to significantly reduce computational time through task and data parallelisation, integrate multiple types of databases within a closed environment, and effectively handle large datasets. These capabilities make it a valuable tool for researchers in the field of bioinformatics. Future research could involve further exploration of the impact of these technologies on bioinformatics software systems, as well as the development of best practices for their implementation in this context.

6.3 Future work

As we look towards the future potential of our bioinformatics microservices system, several exciting opportunities for improvement present themselves. One promising direction for development is the expansion of the system's REST API to consume more tools and APIs, as mentioned by Sheffield et al. [155]. This integration would significantly increase the versatility and functionality of the system and enable it to interact with a wider range of resources and tools.

The experiments conducted in this study focused on evaluating the performance of our proposed framework using various data sets and algorithms. Specifically, we ran an HMM search over the Repbase dataset, a matrix decomposition on genotype expression project data, and time series analysis with convolutional long short-term memory neural networks based on Ustyantsev et al., O'Brien et al. and Shi et al. [156, 163, 170] . There are other areas in which we could potentially extend these experiments. One direction we could take to expand upon our research in bioinformatics to try out different data sets and algorithms to validate the effectiveness of our framework. For example, we could use different a method to identify and mark duplicate reads in high-throughput sequencing data similar to inside Gatk framework or Random Forest as proposed by Chen and Ishwaran to see how well they perform [37, 120]. Ultimately, the goal is to find the reliable Moreover, effective methods for analysing and interpreting biological data and continue refining and improving our framework in the future.

In order to ensure the reliability and generalizability of our framework, we could extend our analysis to examine its robustness under different circumstances [182]. To this end, future research should focus on evaluating the performance of the framework under a variety of conditions using robustness as further factire [98, 113]. This test will allow us to determine the robustness of the framework, and identify any potential weaknesses that need to resolve. Given the the increasing importance of robust architecture in the field of data analysis, such an investigation is essential for advancing our understanding of the framework and improving its effectiveness in practical applications.

Adding network tests to our bioinformatics data framework could help ensure the

network's performance, reliability, and security. These tests involve the evaluation of the network's performance, reliability, security, and compatibility, as well as its ability to handle high levels of usage and traffic. There are various types of network tests that can be performed, including functional testing, performance testing, security testing, compatibility testing, stress testing, load testing, and disaster recovery testing. As Griffeth et al. proposed the use of an experimental method in the testing process and the identification of self-similar structures in networks can improve the efficiency and effectiveness of testing [73]. By conducting these tests, we can better understand how our framework performs under different network conditions and gain insights into its underlying mechanisms, potentially identifying areas for improvement. These efforts may contribute to the development of a more robust and reliable approach for analysing and processing data.

In conclusion, the findings of this study suggest that the implementation of software patterns, microservices, Docker, and advanced software engineering principles and practices can significantly improve the reliability, scalability, performance, efficiency, and productivity of bioinformatics software systems. Future work could involve further exploration of the impact of these technologies on bioinformatics software systems, as well as the development of best practices for their implementation in this context. Additionally, further examination of the robustness, scalability and validity of our framework could help to advance our understanding of the potential applications of our framework and to identify potential directions for future development.

6.4 Conclusion

This study has made significant strides in addressing the research questions surrounding the design, implementation, and optimisation of bioinformatics software systems using software patterns, microservices, Docker, and advanced software engineering principles and practices. Our findings suggest that incorporating these technologies can significantly improve bioinformatics software systems.' reliability, scalability, performance, efficiency, and productivity. In addition, we have identified key considerations and challenges in the design and implementation process, as well

as factors that influence the success of such systems. These insights provide valuable guidance for researchers and practitioners seeking to implement these technologies in developing bioinformatics software systems.

In addition, our research has demonstrated the potential of parallel computation and machine learning techniques to optimise data analysis and integration within our framework. By applying these approaches, we significantly reduced computational time and effectively integrated and analysed data from multiple sources. These capabilities are essential for addressing the challenges of working with large datasets in the field of bioinformatics and for conducting comprehensive analyses.

Overall, our study has contributed valuable insights into the potential benefits and challenges of using software patterns, microservices, Docker, and advanced software engineering principles and practices in the development of bioinformatics software systems. We believe that these findings will be of great value to researchers and practitioners in the field, and we look forward to continuing to explore the potential of these technologies in future work.

Bibliography

[1] Apache spark documentation. https://spark.apache.org/docs/latest/, Nov 2018. Accessed on 2019-01-14.

[2] Cloud vision 2020: The future of the cloud study. https://www.logicmonitor.com/resource/the-future-of-the-cloud-a-cloud-influencers-survey/?utm_medium=pr&utm_source=businesswire&utm_campaign=cloudsurvey, Jan 2018. Accessed: 2020-01-04.

[3] Docker engine overview. https://docs.docker.com/install/, jan 2018. Accessed on 2019-12-07.

[4] Linux vserver. http://linux-vserver.org/, Jul 2018. Accessed on 2020-09-12.

[5] Creating highly available clusters with kubeadm. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/, jan 2019. Accessed on 2019-12-02.

[6] Dockerfile reference. https://docs.docker.com/engine/reference/builder/, jan 2019. Accessed on 2019-11-23.

[7] Swarm mode overview. https://docs.docker.com/engine/swarm/, jan 2019. Accessed on 2019-11-23.

[8] Arvados – open source big data processing and bioinformatics, 2020. Accessed on 2020-01-21.

[9] Openvz. https://openvz.org/, Mar 2021. Accessed on 2021-11-02.

[10] Docker. https://www.docker.com/, Jan 2022. Accessed on 2022-01-02.

[11] Kubernetes. https://kubernetes.io/, Jan 2022. Accessed on 2022-01-02.

[12] Lxc. https://linuxcontainers.org/, Jan 2022. Accessed on 2022-01-02.

[13] Singularity. http://sislabs.io/, Jan 2022. Accessed on 2022-01-12.

[14] M. L. Abbott and M. T. Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley Professional, Upper Saddle River, NJ, 2nd edition, 2015.

[15] M. Abdelhak, S. Grostick, and M. Hanken. *Health Information - E-Book: Management of a Strategic Resource*. Elsevier Health Sciences, 2014.

[16] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, August 1977.

[17] A. Alkhalid, C.-H. Lung, and S. Ajila. Software architecture decomposition using adaptive k-nearest neighbor algorithm. In *2013 26th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, page 1–4, May 2013.

[18] A. Anguera, J. M. Barreiro, J. A. Lara, and D. Lizcano. Applying data mining techniques to medical time series: an empirical case study in electroencephalography and stabilometry. *Computational and structural biotechnology journal*, 14:185–199, May 2016. 27293535[pmid].

[19] C. Avci, B. Tekinerdogan, and I. Athanasiadis. Software architectures for big data: a systematic literature review. *Big Data Analytics*, 5, Aug 2020.

[20] W. Bao, K. Kojima, and O. Kohany. Repbase update, a database of repetitive elements in eukaryotic genomes. *Mobile DNA*, 6, 06 2015.

[21] J. C. Bare and N. S. Baliga. Architecture for interoperable software in biology. *Briefings in Bioinformatics*, 15(4):626–636, Jul 2014.

[22] J. Barker and J. Thornton. Software engineering challenges in bioinformatics. page 12–15, Jan 2004.

[23] C. Barnes, B. Bajracharya, M. Cannalte, Z. Gowani, W. Haley, et al. The biomedical research hub: a federated platform for patient research data. *Journal of the American Medical Informatics Association*, 29(4):619–625, Apr 2022.

[24] R. Batista-Navarro, J. Carter, and S. Ananiadou. Argo: enabling the development of bespoke workflows and services for disease annotation. *Database*, 2016, 05 2016. baw066.

[25] A. Beltre, P. Saha, M. Govindaraju, A. Younge, and R. Grant. Enabling hpc workloads on cloud infrastructure using kubernetes container orchestration mechanisms. 11 2019.

[26] T. Besker, A. Martini, and J. Bosch. Impact of architectural technical debt on daily software development work - a survey of software practitioners. 09 2017.

[27] P. Billingsley. *Probability And Measure, 3rd Ed.* Wiley series in probability and mathematical statistics. Wiley India Pvt. Limited, 2008.

[28] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, 2007.

[29] G. S. Blair and J.-B. Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[30] J. Bosch. Software architecture: The next step. volume 3047, page 194–199, May 2004.

[31] F. Bülthoff and M. Maleshkova. Restful or restless - current state of today's top web apis. *CoRR*, abs/1902.10514, 2019.

[32] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, et al. Apache flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 2015.

[33] V. J. Carey, A. R. Davis, M. F. Lawrence, R. Gentleman, and B. A. Raby. Data structures and algorithms for analysis of genetics of gene expression with bioconductor: Ggtools 3.x. *Bioinformatics*, 25(11):1447–1448, Jun 2009.

[34] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the benefits of mixed data and task parallelism. Jun 1995.

[35] L. Chen, M. Ali Babar, and B. Nuseibeh. Characterizing architecturally significant requirements. *IEEE Software*, 30(2):38–45, 2013.

[36] R. Chen, S. Li, and Z. Li. From monolith to microservices: A dataflow-driven approach. *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, page 466–475, Dec 2017.

[37] X. Chen and H. Ishwaran. Random forests for genomic data analysis. *Genomics*, 99(6):323–329, Jun 2012.

[38] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, et al. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[39] P. Clutterbuck, T. Rowl, and O. Seamons. A case study of sme web application development effectiveness via agile methods. *Electronic Journal of Information Systems Evaluation*, pages 13–26, 2009.

[40] P. Comon. Independent component analysis, a new concept? *Signal Processing*, 36:287–314, Apr 1994.

[41] F. Costa. Big data in biomedicine. *Drug discovery today*, 19, 10 2013.

[42] K. Costello. Gartner forecasts worldwide public cloud revenue to grow 17.5 percent in 2019. https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g, Aug 2019. Accessed: 2020-04-04.

[43] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2. edition edition, Sep 2006.

[44] W. Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, Dec 1992.

[45] J. Darmont, O. Boussaid, J.-C. Ralaivao, and K. Aouiche. An architecture framework for complex data warehouses, 2007.

[46] S. Dash, S. Shakyawar, M. Sharma, and S. Kaushik. Big data in healthcare: management, analysis and future prospects. *Journal of Big Data*, 6, 12 2019.

[47] E. R. Davies. *Computer Vision: Principles, Algorithms, Applications, Learning*. Academic Press, 5th edition edition, Nov 2017.

[48] A. A. C. De Alwis, A. Barros, A. Polyvyanyy, and C. Fidge. *Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems*, volume 11236 of *Lecture Notes in Computer Science*, page 37–53. Springer International Publishing, Cham, 2018.

[49] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, volume 51 of *Osdi'04*, pages 10–10, Berkeley, CA, USA, 01 2004. USENIX Association.

[50] E. Deelman, W. Kaplow, B. Szymanski, P. Tannenbaum, and L. Ziantz. Integrating data and task parallelism in scientific programs. Dec 2002.

[51] P. Desai. A survey of performance comparison between virtual machines and containers. *International Journal Of Computer Sciences And Engineering*, 4:55–59, 07 2016.

[52] K. K. Dey, C. J. Hsiao, and M. Stephens. Visualizing the structure of rna-seq expression data using grade of membership models. *PLoS genetics*, 13(3):e1006599, Mar 2017.

[53] C. Ding, T. Li, and M. Jordan. Convex and semi-nonnegative matrix factorizations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(8):1548–1560, 2011.

[54] C. Ding, T. Li, and M. I. Jordan. Robust nonnegative matrix factorization for data recovery and discovery. *Journal of Machine Learning Research*, 14(1):3413–3459, 2013.

[55] H. Ding, L. Arber, L. Sha, and M. Caccamo. The dependency management framework: a case study of the ion cubesat. In *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, pages 10 pp.–64, 2006.

[56] I. D. Dinov. Volume and value of big healthcare data. *Journal of medical statistics and informatics*, 4:3, 2016. 26998309[pmid].

[57] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[58] D. Eddelbuettel. Parallel computing with r: A brief review. Apr 2020. arXiv:1912.11144 [stat].

[59] L. Elden. *Matrix Methods in Data Mining and Pattern Recognition, Second Edition*. Fundamentals of Algorithms. Society for Industrial and Applied Mathematics, 2019.

[60] P. Emami Khoonsari, P. Moreno, S. Bergmann, J. Burman, M. Capuccini, et al. Interoperable and scalable data analysis with microservices: applications in metabolomics. *Bioinformatics*, 35(19):3752–3760, Oct 2019.

[61] J. Espinosa, S. Kaisler, F. Armour, and W. Money. Big data redux: New issues and challenges moving forward. 01 2019.

[62] F. Esposito. A review on initialization methods for nonnegative matrix factorization: Towards omics data experiments. *Mathematics*, 9(99):1006, Jan 2021.

[63] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[64] R. Finn. Hmmer: Fast and sensitive sequence similarity searches, 2018.

[65] G. Fischer. *Lineare Algebra*. Grundkurs Mathematik : Studium. Vieweg+Teubner Verlag, 2009.

[66] Flexera. State of the cloud report. 01 2019.

[67] M. Fowler and J. Lewis. Microservices. https://martinfowler.com/articles/microservices.html, Mar 2014. Accessed on 2018-11-21.

[68] D. Freedman, R. Pisani, and R. Purves. *Statistics: Fourth International Student Edition*. International student edition. W.W. Norton & Company, 2007.

[69] D. Garlan. *An Introduction to Software Architecture*, page 1–39. Dec 1993. journalAbbreviation: Advances in Software Engineering and Knowledge Engineering 2.

[70] S. Ghosh. *Distributed Systems: An Algorithmic Approach, Second Edition*. Chapman & Hall/CRC Computer and Information Science Series. CRC Press, 2nd edition edition, Jul 2014.

[71] R. Gnanadesikan. *Methods for Statistical Data Analysis of Multivariate Observations*. Wiley Series in Probability and Statistics. Wiley, 2011.

[72] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2013.

[73] N. Griffeth and C. Djouvas. Experimental method for testing networks. page 935–941, Jan 2005.

[74] G. Grimmett, D. Grimmett, and D. Stirzaker. *Probability and Random Processes: Fourth Edition*. Oxford University Press, 2020.

[75] gskchaitanya, J. Creasy, C. Larsen, B. Sigoure, and V. Kiryanov. Opentsdb - the scalable time series database. http://opentsdb.net/overview.html, Dez 2011. Accessed on 2019-02-03.

[76] N. Guan, Z. Du, J. Li, J. Li, and X. Li. Non-negative matrix factorization for analysis of gene expression data. *Bioinformatics*, 26(19):2720–2727, 2010.

[77] J. L. Gustafson. *Amdahl's Law*, pages 53–60. Springer US, Boston, MA, 2011.

[78] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann. Service cutter: A systematic approach to service decomposition. In M. Aiello, E. B. Johnsen, S. Dustdar, and I. Georgievski, editors, *Service-Oriented and Cloud Computing*, Lecture Notes in Computer Science, page 185–200, Cham, 2016. Springer International Publishing.

[79] J. Hamilton. On designing and deploying internet-scale services. page 231–242, Jan 2007.

[80] A. P. Heath, V. Ferretti, S. Agrawal, M. An, J. C. Angelakos, et al. The nci genomic data commons. *Nature Genetics*, 53(33):257–262, Mar 2021.

[81] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in cloud computing: What it is, and what it is not. page 23–27, 2013.

[82] N. Higham. *Functions of Matrices: Theory and Computation*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2008.

[83] O. Hummel, H. Eichelberger, A. Giloj, D. Werle, and K. Schmid. A collection of software engineering challenges for big data system development. page 362–369, Aug 2018.

[84] A. Hyvärinen, J. Karhunen, and E. Oja. *Independent Component Analysis*. John Wiley & Sons, Jun 2001. Google-Books-ID: 9TQNEAAAQBAJ.

[85] International Organization for Standardization. Information technology – open distributed processing – reference model: Architecture iso/iec 10746-3:2009. *Iso*, 2019.

[86] R. Jain and N. Chandhok. Web distribution systems : Caching and replication. https://www.cse.wustl.edu/~jain/cis788-99/ftp/web_caching/index.html, Dec 1999. Accessed on 2021-12-21.

[87] V. Jalili, E. Afgan, Q. Gu, D. Clements, D. Blankenberg, et al. The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2020 update. *Nucleic Acids Research*, 48(W1):W395–w402, Jul 2020.

[88] M. Kassab, M. Mazzara, J. Lee, and G. Succi. Software architectural patterns in practice: an empirical study. *Innovations in Systems and Software Engineering*, 14, Dec 2018.

[89] P. Kaur, A. Singh, and I. Chana. Computational techniques and tools for omics data analysis: State-of-the-art, challenges, and future directions. *Archives of Computational Methods in Engineering*, 28:1–37, Feb 2021.

[90] G. Kecskemeti, A. Marosi, and A. Kertész. *The ENTICE approach to decompose monolithic services into microservices*. Ieee, Innsbruck, Austria, Jul 2016.

[91] C. E. Kennedy and J. P. Turley. Time series analysis as input for clinical predictive modeling: Modeling cardiac arrest in a pediatric icu. *Theoretical Biology and Medical Modelling*, 8(1):40, Oct 2011.

[92] J. Kopecký, P. Fremantle, and R. Boakes. A history and future of web apis. *it - Information Technology*, 56, 01 2014.

[93] J. Koster and S. Rahmann. Snakemake–a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, oct 2012.

[94] S. Krakowiak. Middleware architecture with patterns and frameworks. *Creative Commons*, Feb 2019.

[95] P. Kruchten. Architectural blueprints: The 4+1 view model of software architecture. *IEEE Software*, 12(6):42–50, Nov 1995. arXiv: 2006.04975.

[96] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, Usa, 1 edition, 2008.

[97] P. Kulkarni and P. Frommolt. Challenges in the setup of large-scale next-generation sequencing analysis workflows. *Computational and structural biotechnology journal*, 15:471–477, Oct 2017. 29158876[pmid].

[98] N. Laranjeiro, J. a. Agnelo, and J. Bernardino. A systematic review on software robustness assessment, May 2022.

[99] J. Lawless. *Statistical Models and Methods for Lifetime Data*. Wiley Series in Probability and Statistics. Wiley, 2011.

[100] B. Lawlor and P. Walsh. Engineering bioinformatics: building reliability, performance and productivity into bioinformatics software. *Bioengineered*, 6(4):193–203, May 2015.

[101] D. Lay. *Linear Algebra and Its Applications*. Addison-Wesley, 2012.

[102] D. D. Lee, M. N. Ho, and H. S. Seung. Image interpolation using sparse matrix factorization. *IEEE Transactions on Image Processing*, 11(8):1298–1308, 2002.

[103] D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(67556755):788–791, Oct 1999.

[104] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In *Advances in Neural Information Processing Systems*, pages 556–562, 2001.

[105] T. J. Lee, Y. Pouliot, V. Wagner, P. Gupta, D. W. Stringer-Calvert, et al. Biowarehouse: a bioinformatics database warehouse toolkit. *BMC Bioinformatics*, 7(1):170, Mar 2006.

[106] J. Leipzig. A review of bioinformatic pipeline frameworks. *Brief Bioinform*, 18(3):530–536, May 2017.

[107] A. Lesk. *Introduction to Bioinformatics*. Oxford University Press, May 2019.

[108] M. A. Levinson, J. Niestroy, S. Al Manir, K. Fairchild, D. E. Lake, et al. Fairscape: a framework for fair and reproducible biomedical analytics. *Neuroinformatics*, 20(1):187–202, Jan 2022.

[109] J. Li, J. Zhang, X. Li, X. Li, and X. Li. Non-negative matrix factorization-based classification of microarray data by using the global structure of data. *Journal of bioinformatics and computational biology*, 11(6):1350017, 2013.

[110] C.-J. Lin, Y.-Y. Chen, C.-F. Pan, V. Wu, and C.-J. Wu. Dataset supporting blood pressure prediction for the management of chronic hemodialysis. *Scientific data*, 6(1):313–313, Dec 2019. 31819065[pmid].

[111] B. H. Liskov. A design methodology for reliable software systems. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part I*, AFIPS '72 (Fall, part I), page 191–199, New York, NY, USA, Dec 1972. Association for Computing Machinery.

[112] M. Lovino, V. Randazzo, G. Ciravegna, P. Barbiero, E. Ficarra, et al. A survey on data integration for multi-omics sample clustering. *Neurocomputing*, 488:494–508, Jun 2022.

[113] C.-H. Lung and K. Kalaichelvan. An approach to quantitative software architecture sensitivity analysis. *International Journal of Software Engineering and Knowledge Engineering*, 10(01):97–114, Feb 2000.

[114] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1st edition edition, Mar 1996.

[115] M. Maleshkova, C. Pedrinaci, and J. Domingue. Investigating web apis on the world wide web. *Proceedings - 8th IEEE European Conference on Web Services, ECOWS 2010*, 12 2010.

[116] G. Marcais, B. Solomon, R. Patro, and C. Kingsford. Sketching and sublinear data structures in genomics. *Annual Review of Biomedical Data Science*, 2:1–26, Apr 2019.

[117] R. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 1st edition edition, Jul 2013.

[118] D. Mattioli and A. Tilley. Amazon has long ruled the cloud. now it must fend off rivals.

[119] M. Mattox. *Rancher Deep Dive: Manage enterprise Kubernetes seamlessly with Rancher*. Packt Publishing, 2022.

[120] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, et al. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Res.*, 20(9):1297–1303, Sep 2010.

[121] X. Min, W. Zeng, N. Chen, T. Chen, and R. Jiang. Chromatin accessibility prediction via convolutional long short-term memory networks with k-mer embedding. *Bioinformatics (Oxford, England)*, 33(14):i92–i101, Jul 2017.

[122] B. B. Misra, C. D. Langefeld, M. Olivier, and L. A. Cox. Integrated omics: Tools, advances, and future approaches. *Journal of Molecular Endocrinology*, pages Jme–18–0055, Jul 2018.

[123] F. Mörchen, A. Ultsch, and O. Hoos. Extracting interpretable muscle activation patterns with time series knowledge mining. *KES Journal*, 9:197–208, Sep 2005.

[124] R. Nagarajan, M. Ahmed, and A. Phatak. Database challenges in the integration of biomedical data sets. Oct 2004.

[125] D. Namiot and M. sneps sneppe. On micro-services architecture. *Interenational Journal of Open Information Technologies*, 2:24–27, 09 2014.

[126] B. C. Neuman. Scale in distributed systems. pages 463–489, 1994.

[127] S. Newman. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Incorporated, 2019.

[128] J. Nickoloff. *Docker in Action*. Manning Publications Co., Usa, 1st edition, 2016.

[129] J. A. Novella, P. Emami Khoonsari, S. Herman, D. Whitenack, M. Capuccini, et al. Container-based bioinformatics with pachyderm. *Bioinformatics*, 35(5):839–846, Mar 2019.

[130] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. *Usenix*, page 305–320, Jan 2014.

[131] M. Onken, M. Eichelberg, J. Riesmeier, and P. Jensch. Digital imaging and communications in medicine. page 427–454, 2011.

[132] S. Panda, S. Mondal, R. Dewri, and A. K. Das. Towards achieving efficient access control of medical data with both forward and backward secrecy. *Computer Communications*, 189:36–52, May 2022.

[133] D. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, Se-5(2):128–138, Mar 1979.

[134] D. Patterson, A. Brown, P. Broadwell, G. C, M. Chen, et al. Recovery oriented computing (roc): Motivation, definition, techniques, and case studies. Apr 2002.

[135] C. Pautasso and E. Wilde. *Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design*. Jan 2009. journalAbbreviation: 18th International World Wide Web Conference.

[136] W. R. Pearson. Finding protein and nucleotide similarities with fasta. *Current protocols in bioinformatics*, 53:3.9.1–3.925, Mar 2016.

[137] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, Sep 2000.

[138] A. Periasamy, G. Kapoor, and Harshavardhana. Minio object storage - open source, s3 compatible, enterprise hardened and really, really fast. https://min.io/product/overview, Nov 2014. Accessed on 2019-01-21.

[139] K. Peters, J. Bradbury, S. Bergmann, M. Capuccini, M. Cascante, et al. Phenomenal: processing and analysis of metabolomics data in the cloud. *GigaScience*, 8(2):giy149, Feb 2019.

[140] C. Ramamoorthy and B. Wah. Knowledge and data engineering. *IEEE Transactions on Knowledge and Data Engineering*, 1:9–16, Mar 1989.

[141] G. Recht. *Bundesdatenschutzgesetz (BDSG)* -. G. Recht, 2014.

[142] M. Reddy. *API Design for C++*. Elsevier Science, 2011.

[143] H. L. Rehm, A. J. H. Page, L. Smith, J. B. Adams, G. Alterovitz, et al. Ga4gh: International policies and standards for data sharing across genomic research and healthcare. *Cell Genomics*, 1(2):100029, Nov 2021.

[144] M. Reisman. Ehrs: The challenge of making electronic data usable and interoperable. *P & T : a peer-reviewed journal for formulary management*, 42(9):572–575, Sep 2017. 28890644[pmid].

[145] D. Renzel, P. Schlebusch, and R. Klamma. Today's top "restful" services and why they are not restful. In X. S. Wang, I. Cruz, A. Delis, and G. Huang, editors, *Web Information Systems Engineering - WISE 2012*, pages 354–367, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[146] G. V. Research. Application container market analysis report by deployment, by platform (kubernetes, docker), by organization size (smes, large enterprise), by service, by application, by region, and segment forecasts, 2019 - 2025. https://www.grandviewresearch.com/industry-analysis/application-container-market, Feb 2019. Accessed on 2020-01-21.

[147] C. Richardson. *Microservices Patterns: With examples in Java*. Manning, Shelter Island, New York, 1st edition edition, Nov 2018.

[148] E. M. Robertsen, T. Kahlke, I. A. Raknes, E. Pedersen, E. Kjærner-Semb, et al. Meta-pipe - pipeline annotation, analysis and visualization of marine metagenomic sequence data. *CoRR*, abs/1604.04103, 2016.

[149] A. Roy, Y. Diao, U. Evani, A. Abhyankar, C. Howarth, et al. Massively parallel processing of whole genome sequence data: An in-depth performance study. In *Proceedings of the 2017 ACM International Conference on Management of Data*, Sigmod '17, page 187–202, New York, NY, USA, 2017. Association for Computing Machinery.

[150] J. Saltz, A. Sharma, G. Iyer, E. Bremer, F. Wang, et al. A containerized software system for generation, management, and exploration of features from whole slide tissue images. *Cancer research*, 77(21):e79–e82, Nov 2017.

[151] J. Schellenberger and B. Palsson. Use of randomized sampling for analysis of metabolic networks. *Journal of Biological Chemistry*, 284(9):5457–5461, Feb 2009.

[152] G. Schryen. Parallel computational optimization in operations research: A new integrative framework, literature review and research directions. (arXiv:1910.03028), Oct 2019. arXiv:1910.03028 [cs].

[153] A. Seffah, M. Donyaee, R. Kline, and H. Padda. Usability measurement and metrics: A consolidated model. *Software Quality Journal*, 14:159–178, Jun 2006.

[154] S. P. Shah, Y. Huang, T. Xu, M. M. Yuen, J. Ling, et al. Atlas – a data warehouse for integrative bioinformatics. *BMC Bioinformatics*, 6(1):34, Feb 2005.

[155] N. C. Sheffield, V. R. Bonazzi, P. E. Bourne, T. Burdett, T. Clark, et al. From biomedical cloud platforms to microservices: next steps in fair data and analysis. *Scientific Data*, 9(11):553, Sep 2022.

[156] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W. kin Wong, et al. Convolutional lstm network: A machine learning approach for precipitation nowcasting, 2015.

[157] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, page 1–10, Incline Village, NV, USA, May 2010. Ieee.

[158] J. A. M. Sidey-Gibbons and C. J. Sidey-Gibbons. Machine learning in medicine: a practical introduction. *BMC Medical Research Methodology*, 19(1):64, Mar 2019.

[159] L. B. Silva, R. C. Jimenez, N. Blomberg, and J. Luis Oliveira. General guidelines for biomedical software development. *F1000Research*, 6:273, Jul 2017.

[160] R. Sokal and F. Rohlf. Biometry: the principles and practice of statistics in biological research / robert r. sokal and f. james rohlf. *SERBIULA (sistema Librum 2.0)*, Apr 2013.

[161] R. M. Stallman and R. McGrath. Gnu make - a program for directing recompilation, 1991.

[162] M. v. Steen and A. S. Tanenbaum. *Distributed Systems*. CreateSpace Independent Publishing Platform, 3rd edition edition, Feb 2017.

[163] G. L. Stein-O'Brien, R. Arora, A. C. Culhane, A. V. Favorov, L. X. Garmire, et al. Enter the matrix: Factorization uncovers knowledge from omics. *Trends in Genetics*, 34(10):790–805, Oct 2018.

[164] M. Stolarczyk, V. Reuter, J. Smith, N. Magee, and N. Sheffield. Refgenie: a reference genome resource manager. *GigaScience*, 9, Feb 2020.

[165] M. Stolarczyk, B. Xue, and N. Sheffield. Identity and compatibility of reference genome resources. *NAR Genomics and Bioinformatics*, 3, Apr 2021.

[166] G. Strang. *Linear algebra and its applications*. Thomson, Brooks/Cole, Belmont, CA, 2006.

[167] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2014.

[168] D. Taibi and K. Systä. From monolithic systems to microservices: A decomposition framework based on process mining. 05 2019.

[169] J. Tzeng, H. H.-S. Lu, and W.-H. Li. Multidimensional scaling for large genomic data sets. *BMC Bioinformatics*, 9(1):179, Apr 2008.

[170] K. Ustyantsev, A. Blinov, and G. Smyshlyaev. Convergence of retrotransposons in oomycetes and plants. In *Mobile DNA*, 2017.

[171] R. Van Der Straeten, T. Mens, and S. Baelen. *Challenges in Model-Driven Software Engineering*, volume 5421. Sep 2008. journalAbbreviation: MODELS 2008. LNCS.

[172] M. van Steen and A. S. Tanenbaum. A brief introduction to distributed systems. *Computing*, 98(10):967–1009, Oct 2016.

[173] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek. Kubernetes as an availability manager for microservice applications. *CoRR*, abs/1901.04946, 2019.

[174] M. A. Vega-Rodríguez and J. M. Granado-Criado. Parallel programming in bioinformatics: Some interesting approaches. *International Journal of Parallel Programming*, 47(2):293–295, Apr 2019.

[175] G. Vera, R. C. Jansen, and R. L. Suppi. R/parallel – speeding up bioinformatics analysis with r. *BMC Bioinformatics*, 9(1):390, Sep 2008.

[176] J. Vivian, A. A. Rao, F. A. Nothaft, C. Ketchum, J. Armstrong, et al. Toil enables reproducible, open source, big biomedical data analyses. *Nature Biotechnology*, 35(4):314–316, Apr. 2017.

[177] T. Vresk and I. Čavrak. Architecture of an interoperable iot platform based on microservices. In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, page 1196–1201, May 2016.

[178] M. Štefanko, O. Chaloupka, and B. Rossi. *The Saga Pattern in a Reactive Microservices Environment*. Jan 2019.

[179] F. Waas, R. Wrembel, T. Freudenreich, M. Thiele, C. Koncilia, et al. On-demand elt architecture for right-time bi: Extending the vision. *Int. J. Data Warehous. Min.*, 9(2):21–38, Apr. 2013.

[180] S. Weerasinghe, R. Kathriarachchi, B. Hettige, and A. Karunananda. Resource sharing in distributed environment using multi-agent technology. *International Journal of Computer Applications*, 167:28–32, Jun 2017.

[181] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 4th edition, 2015.

[182] A. Yang, M. Troup, and J. Ho. Scalability and validation of big data bioinformatics software. *Computational and Structural Biotechnology Journal*, 15, Jul 2017.

[183] F. Yao, J. Coquery, and K.-A. Lê Cao. Independent principal component analysis for biologically meaningful dimension reduction of large biological data sets. *BMC Bioinformatics*, 13(1):24, Feb 2012.

[184] A. D. Yates, J. Adams, S. Chaturvedi, R. M. Davies, M. Laird, et al. Refget: standardized access to reference sequences. *Bioinformatics*, 38(1):299–300, Jan 2022.

[185] S. Yau and J. Collofello. Design stability measures for software maintenance. *IEEE Transactions on Software Engineering*, Se-11(9):849–856, Sep 1985.

[186] S. S. Yau and J. S. Collofello. Some stability measures for software maintenance. *Ieee Transactions On Software Engineering*, (6):8, 1980.

[187] S. L. Zeger, R. Irizarry, and R. D. Peng. On time series analysis of public health and biomedical data. *Annual Review of Public Health*, 27:57–79, 2006.

[188] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: State-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, May 2010.

[189] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, et al. A comparative study of containers and virtual machines in big data environment. (arXiv:1807.01842), Jul 2018. arXiv:1807.01842 [cs].

[190] Y. Zhao and Y. Liu. Oclstm: Optimized convolutional and long short-term memory neural network model for protein secondary structure prediction. *PLoS ONE*, 16(2):e0245982, Feb 2021.

[191] O. Zimmermann. Microservices tenets. *Computer Science - Research and Development*, 32(3):301–310, Jul 2017.

Appendix A

Appendix

A.1 Mathematical background

A.1.1 Matrix inverses

Indeed, since $\mathbb{R}^{n\times m},$ with $n, m \in \mathbb{N}$ is only a ring, the existence of inverses might be absent. Moreover, the rank of the matrix might influence its existence. We assume that the data is preprocessed, that we can assume the input data X has full rank. Concerning the inverse, we can use simple mathematical properties to create the left and right inverse of a matrix. However, this fact relies on two properties .

$$(C^T C)^{-1}(C^T C) = I \tag{A.1}$$

where $C \in \mathbb{R}^{n\times m},$ with $n, m \in \mathbb{N},$ $\text{rank}(C) = \min\{m, n\}$, I being the identity matrix and ^{T} the tranpose operator.

$$(DD^T)(DD^T)^{-1} = I \tag{A.2}$$

where $D \in \mathbb{R}^{n\times m},$ with $n, m \in \mathbb{N},$ $\text{rank}(D) = \min\{m, n\}$. The term $(C^T C)^{-1}C^T$ and $D^T(DD^T)^{-1}$ are referred to as left respectively right inverse [65,101,166]. Therefore, the problem of low-dimensional representation has an equivalent matrix transformation [59, 82]. Depending on the matrix decomposition approach, either the problem of equations 4.1 and 4.2 are equivalent problems.

A.1.2 Principal component analysis

Concerning linear algebra, the main theorem for principal component analysisis the spectral theorem [72,166]. This fact proves the existence of orthogonal eigenvectors.

Theorem A.1.1. *(Real spectral theorem) Let $A \in \mathbb{R}^{m\times n}$. If A is symmetric ($A^T = A$), then A is orthogonal diagonalizable and has real eigenvalues. In other words, there exists a decomposition.*

$$A = U\Lambda U^T \tag{A.3}$$

with

$$U = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \dots & u_n \\ | & | & & | \end{bmatrix}$$

being orthogonal ($U^T U = UU^T = I_n$) matrix. The columns of matrix U are the eigenvectors of A. In addition, the eigenvalues are a diagonal matrix with eigenvalues of A

$$\Lambda = diag(\lambda_1, \lambda_2, \ldots, \lambda_n) = \begin{bmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_n \end{bmatrix}$$

Concretely, for the eigenvalues $\lambda_1, \ldots, \lambda_n \in \mathbb{R}$ *and eigenvectors* $v_1, \ldots, v_n \in \mathbb{R}^m$ *are defined as*

$$Av_i = \lambda_i v_i$$

the that for $i \in \{1, 2, \ldots n\}$*. Moreover, the eigenvectors are the orthogonal and non-zero.*

The following corollary and proposition allow proving helpful properties concerning AA^T and A^TA.

Corollary A.1.1. *If* $A \in \mathbb{R}^{m \times n}$ *then* $AA^T \in \mathbb{R}^{m \times m}$ *and* $A^TA \in \mathbb{R}^{n \times n}$ *are both symmetric.*

One interesting observation that follows from this corollary, is that the symmetric property of the matrices AA^T and A^TA implies that their eigenvectors form a complete orthonormal system.

Proposition A.1.1. *The matrices* AA^T *and* A^TA *share the same non-zero and are positive eigenvalues.*

In summary, these results provide important insights into the properties of matrices that are crucial for understanding and applying principal component analysis. These results are not only limited to principal component analysisbut also have broader implications in the field of linear algebra and matrix theory, providing a deeper understanding of the properties of matrices and their eigenvalues.

In this section, we will delve into the exciting field of property theory and statistics. Of particular relevance to our study is the area of probability theory We negin with the concept of multi-dimensional random variables [27, 74]. This concept forms the foundation of our analysis and will be the focus of the upcoming discussion.

Definition A.1.1. *Let* $(\Omega, \mathcal{A}, P)$ *be a probability space and* $m \in \mathbb{N}$*, with* $m > 1$*. Moreover, let* $\mathcal{B}(\cdot)$ *be the Borel* σ*-Algebra. Then a multi-dimensional random variable is a map*

$$X : (\Omega, \mathcal{A}, P) \to (\mathbb{R}^m, \mathcal{B}(\mathbb{R}^m)) \text{ with } X^{-1}(\mathcal{B}(\mathbb{R}^m)) \subset \mathcal{A}$$

Remark. *There is another equivalent definition for the multi-dimensional random variable. Let* $X = (X_1, X_2, \ldots, X_m)$ *with* $X_1, X_2, \ldots, X_m$ *being real random variables. Then* $X = (X_1, X_2, \ldots, X_m)$ *is a multi-dimensional random variable over* $(\Omega, \mathcal{A}, P)$ *because a map to* $\mathbb{R}^m$ *is measurable if its component maps are measurable.*

Further, a multi-dimensional random variable is also referred to as a random vector.

In this section on matrix decomposition, when considering random variables X we only consider discrete random variables i.e. they are only defined on countable sets. Concretely, we consider the sample as countable events.Furthermore, we require the definition of the expected value of an multi-dimensional random variable.

Definition A.1.2. *Let $X = (X_1, X_2, \ldots, X_m)$ be a multi-dimensional random variable. The expected value of the multi-dimensional random variableX is defined as*

$$E(X) = (E(X_1), E(X_2), \ldots, E(X_m))$$

Remark. *Generally, the expected value of X is also reffered to as $\mu = E(X)$ with the components $\mu = (\mu_1, \ldots, \mu_m)$ and $\mu_i = E(X_i)$. This definition can then also be exspanded to a matrix of random variables $X = (X_{ij})$ with $i \in \{1, \ldots, m\}; j \in \{1, \ldots, n\}; m, n \in \mathbb{N}$. For a discrete random variable X the expected value is defined as*

$$E(X) = \sum_{i \in I} x_i p_i = \sum_{i \in I} x_i P(X = x_i)$$

where $(x_i)_{i \in I}$ the values of X with $(p_i)_{i \in I}$ the related probability and I a countable index set.

Let $X = (X_1, \ldots, X_n)$ be n multi-dimensional random variable with each discrete random variable X_i over the $\mathbb{R}^m$. Therefore, each X_i can be seen as element in $\mathbb{R}^m$. We denote the associated column vector as $x_i \in \mathbb{R}^m$ in a matrix.

$$X = \begin{pmatrix} | & | & & | \\ x_1 & x_2 & \ldots & x_n \\ | & | & & | \end{pmatrix} \tag{A.4}$$

Further we use discrete uniform distribution as the probability measure p_i i.e.

$$P(X = x) = \begin{cases} \frac{1}{n}, & \text{for } x = x_i;\ i \in \{1, 2, \ldots, n\} \\ 0, & \text{otherwise} \end{cases}$$

Then we the expected value is defined as

$$E(X) = \mu_X = \frac{1}{n} \left(\sum_{i=1}^{n} x_i \right) \in \mathbb{R}^m \tag{A.5}$$

After establishing these fundamental concepts, we now proceed to define the concept of covariance in our analysis.

Definition A.1.3. *Let $X = (X_1, \ldots, X_n)$ be a multi-dimensional random variable-with components as vector. Then we define the covariance as*

$$\sigma_{ij} = \mathrm{Cov}(X_i, X_j) = E\left[(X_i - \mu_i)(X_j - \mu_j)\right]$$

Further the covariance as matrix is defined as

$$\Sigma = \begin{pmatrix} \sigma_{11} & \ldots & \sigma_{1n} \\ \vdots & \ddots & \vdots \\ \sigma_{n1} & \ldots & \sigma_{nn} \end{pmatrix}$$

Remark. *The diagonal of Σ is the variance*

$$\sigma_{ii} = E(X_i - \sigma_i)^2 = \mathit{Var}(X_i) = \sigma_i^2$$

The matrix Σ is symmetric since $\sigma_{ij} = \sigma_{ji}$. When using a matrix notation we can write the covariance as

$$\Sigma = \mathrm{Cov}(X) = E[(X - \mu)(X - \mu)^T]$$

In statistics, the covariance of a matrix $X = (X_{jk})_{j,k \in 1,\ldots,n}$ is has some details have to be outlines. the covariance of a matrix is a measure of how much two variables change together. It is defined as the average product of the deviations of the elements of the two matrices from their respective means [71]. The formula for the covariance of a matrix with n elements is:

$$\mathrm{Cov}(X_{jk}) = \frac{1}{n-1} \sum_{i=0}^{n} \left[(X_{ij} - \mu_{X_j})(X_{jk} - \mu_{X_j})\right]$$

This formula is used to calculate the sample covariance, which is an estimate of the population covariance. The reason that the denominator is n-1 instead of n is because it is an unbiased estimate of the variance [160]. When you use the sample mean to estimate the population mean, you are introducing a small amount of error. Dividing by n-1 instead of n reduces this error, making it a more accurate estimate of the population variance [68].

In our study, we consider only samples so we need to add the definition statistical sample covariance.

Definition A.1.4. *Let matrix $A \in \mathbb{R}^{m \times m}$ with A_μ defined as*

$$A_\mu = \begin{bmatrix} | & | & & | \\ a_1 - \mu_A & x_2 - \mu_A & \ldots & x_n - \mu_A \\ | & | & & | \end{bmatrix} \tag{A.6}$$

The sample covariance matrix of A is defined as

$$\begin{aligned} Cov(A) &= \frac{1}{n-1} A_\mu A_\mu^T \\ &= \frac{1}{n-1} \sum_{i=1}^{n} \langle a_i - \mu_A, a_i - \mu_A \rangle \\ &= \frac{1}{n-1} \sum_{i=1}^{n} (a_i - \mu_A)^2 \end{aligned} \tag{A.7}$$

With these statistical results we can focus on the key results needed for principal component analysis.

Theorem A.1.2. *(Singular value decomposition theorem) Let a $A \in \mathbb{R}^{m \times n}$. The matrix A can be decomposition into*

$$A = U \tilde{S} V^T \tag{A.8}$$

with

$$\tilde{S} = \begin{pmatrix} S & 0 \\ 0 & 0 \end{pmatrix}$$

and $U \in \mathbb{R}^{m \times m}$, $V \in \mathbb{R}^{n \times n}$ are both orthogonal matrices and the matrix S is diagonal:

$$S = diag(\sigma_1, \sigma_2, \ldots, \sigma_r) = \begin{bmatrix} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_r \end{bmatrix}$$

where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$ are unique Furthermore, it referred to the singular values of A. Moreover, the rank of A is defined as $r = rank(A) \leq \min(m, n)$. The triple $(U, \tilde{S}, V$ is called the singular value decomposition of A.

So the decomposition of a matrix $A_\mu \in \mathbb{R}^{m \times n}$ as defined in equation A.6. With real spectrum theorem A.1.1, we know there is a decomposition for the covariance matrix of A_μ as

$$\mathrm{Cov}(A_\mu) = U \Lambda U^T = \frac{1}{n-1} A_\mu A_\mu^T \tag{A.9}$$

Then the columns of the matrix $U^T X$ correspond to the principal components. With the singular value decomposition theorem A.1.2 we have a decomposition for the matrix A_μ as

$$A_\mu = U \tilde{S} V^T \tag{A.10}$$

When we combine equation A.9 and A.10 we get

$$\mathrm{Cov}(A_\mu) = \frac{1}{n-1} U\tilde{S}V^T \cdot V\tilde{S}U^T = U\frac{\tilde{S}^2}{n-1}U^T \tag{A.11}$$

Therefore, the eigenvalues of $\mathrm{Cov}(A_\mu)$ and the diagonal matrix of the singular value decompositionare related as follows

$$\lambda_i = \frac{s_i^2}{n-1},\ i \in \{1, 2, \ldots, n\}$$

A.1.3 Independent component analysis

Let S be a multi-dimensional random variable with statistically independent random variables components (as defined in definition A.1.1) For independent component analysis to be applied, at most one of the random variables components must be Gaussian distributed. When we refer to the multiplication of the matrix with a multi-dimensional random variable we refer to a matrix multiplied with an element of the target space in $\mathbb{R}^m$. The random variables are multiplied by a matrix A — also referred to mixing matrix. For simplicity, this mixing matrix is assumed to be quadratic. The result is mixed multi-dimensional random variable X, which has the same dimension as S.

$$X = AS$$

The goal of ICA is to reconstruct the independent random variables in the vector S as faithfully as possible. For this, only the result of the mixture X is available, and the knowledge that the random variables were originally stochastically independent. A suitable matrix $B = A^{-1}$ is searched, so that

$$S = A^{-1}X$$

Without the knowledge of A, we have to calculate B differently. Generally, the components of X might correlate with each other. Therefore, we can choose B with principal component analysis in such a way that these correlations disappear. However, while principal component analysis uses covariance, independent component analysis uses other methods to calculate the components. Thus, independent component analysisremoves the correlations and makes the components stochastically independent from each other.

We will now outline some of the critical results that facilitate independent component analysis. The method independent component analysis can be viewed as a matrix decomposition that relies mainly on probability results. core diffrences to PCA

Definition A.1.5. *(Independent component analysis) Let X be a multi-dimensional random variable over a probability space (Ω, Σ, P) with $X : \Omega \to \mathbb{R}^n$. Further, let covariance $\mathrm{Cov}(X)$ be finite. We call the tuple of matrices. (F, Λ) independent component analysis of X if*

1. *the covariance $\mathrm{Cov}(X)$ can be decomposed into*

$$\mathrm{Cov}(X) = U\Lambda^2 U^T$$

   where Λ is a diagonal real postive matrix and F has full column rank p. (compare to the sectral theorem A.1.1)
2. *the multi-dimensional random variable X can be written as*

$$X = \Lambda Y$$

   with Y is a multi-dimensional random variable $Y : \Omega \to \mathbb{R}^p$ and $\mathrm{Cov}(Y) = \Lambda^2$.
3. *The columns of U have unit norm*
4. *The entries in Λ are sorted in decreasing order*
5. *The entries of the largest norm in each column of U are positive*

Remark. *The last three properties allow uniqueness. When considering multi-dimensional random variable we have to consider that the statistical independence is unaffected by the scalar multiplication of an element of the target space in $\mathbb{R}^m$ or a permutation of the components. Therefore, we are considering the realation between the independent component analysis.*

Property A.1.1. *We have the following relation for the independent component analysis over a multi-dimensional random variable X for (F, Λ) and (F', Λ')*

$$(F, \Lambda) \sim (F', \Lambda') \text{ with } F' = F\bar{M}DP \text{ and } \Lambda' = P^T\bar{M}\Lambda P$$

where $\bar{M} \in \mathbb{R}^{p\times p}$ is a invertible diagonal positive scaling matrix, $D \in \mathbb{R}^{p\times p}$ a diagonal matrix with entries unit norm and $P \in \mathbb{R}^{p\times p}$ a permutation matrix.

There is one important condition attached to the applicability of the independent component analysis. The independent components must not be normally distributed — at most, one component. In classical probability theory, we usually take random variables to be normally distributed. Variables are usually assumed to be normally distributed. Suppose the matrix A is orthogonal, and the S_i are normally distributed. Then also, the X_i are typically distributed and independent. The following theorem summarizes the property

Theorem A.1.3. *Let X be a multi-dimensional random variable with independent components, of which most one is gaussian. Further, the densities are not reduced to point-like mass. Let $C \in \mathbb{R}^{m\times m}$ and X, S multi-dimensional random variable with dimension m. Then the following three points are equivalent*

1. *The components X_i are pairwise independent.*
2. *The components X_i are mutually independent.*
3. *We have the following property*

$$C = \Lambda P$$

where Λ is diagonal and P is a permutation matrix.

For our experiments, we used the fastICA algorithm to determine the copmponets. Here we outline the steps of the algorithm.

Algorithm 1 fastICA

Input c number of required components,

Input $X \in \mathbb{R}^{m \times n}$ prewhitened matrix, which refers to principal component analysis on the covariance of X where each column represents an m-dimensional sample with $c \leq m$

for $p \in \{1, \dots c\}$ **do**
  $w_p \leftarrow$ multi-dimensional random variable of dimension m
  while w_p changes **do**
    $w_p \leftarrow \frac{1}{n} X g(w_p^T X)^T - \frac{1}{n} g'(w_p^T X) 1_n w_p$
    $w_p \leftarrow w_p - \sum_{j=1}^{p-1} (w_p^T w_j) w_j$
    $w_p \leftarrow \frac{w_p}{\|x_p\|}$
  end while
end for
$W \leftarrow [w_1, \dots w_c]$
$S \leftarrow W^T X$

Output $W \in \mathbb{R}^{m \times c}$ Un-mixing matrix where each column projects X onto independent component

Output $S \in \mathbb{R}^{c \times n}$ independent component matrix, with n columns representing a sample with c dimension

Let 1_n be a column vector of 1's of dimension n. Using Hyvärinen to extract multiple components with a nonquadratic nonlinear function

$$f(u) = \log(\cosh(u)), \quad g(u) = \tanh(u), \quad g'(u) = 1 - \tanh^2(u)$$

A.1.4 Non-negative matrix factorisation

Let $X \in \mathbb{R}^{m \times n}$ with nonnegative entries. Furthermore, let the number of components $k \in \mathbb{N}$ which is smaller than m and than n. The nonnegative matrix factorization

then consists of a matrix $W \in \mathbb{R}^{m \times k}$ and a matrix $H \in \mathbb{R}^{k \times n}$, both of which have nonnegative entries and minimizes the Frobenius norm of the difference

$$\min_{W,J} \|X - WH\|_F \tag{A.12}$$

where Frobenius norm $\|\cdot\|_F$ with $A \in \mathbb{R}^{m \times n}$ is defined as

$$\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$$

This definition, except for the requirement of non-negativity, corresponds exactly to that of principal component analysis. The factorization is not uniquely determined by the minimization problem from equation A.12. For example, for a permutation matrix $\Pi \in S_k$, the matrices $W' = W\Pi$ and $H' = \Pi^{-1}H$ are also minimizers of $\|X - W'H'\|_F$, so the order of the factors is thus completely indeterminate. The scaling of the factors can also vary.

Lee and Seung gave the following multiplicative update rule for determining W and H [103]:

$$W \leftarrow W \odot \frac{W^T X}{W^T W H} \tag{A.13}$$

$$H \leftarrow H \odot \frac{X H^T}{W H H^T} \tag{A.14}$$

Here, $\odot$ denotes the Hadamard product, i.e., element-wise multiplication. Let $A = (a_{ij}), B = (b_{ij}) \in \mathbb{R}^{m \times n})$ then the Hadamard product of A, B is

$$A \odot B = (a_{ij} \cdot b_{ij}) = \begin{pmatrix} a_{11} \cdot b_{11} & \cdots & a_{1n} \cdot b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} \cdot b_{m1} & \cdots & a_{mn} \cdot b_{nm} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

The numerator and denominator in each entry should also be divided for the fractions. These update rules can be derived from the gradient descent with the addition of special prefactors. The advantage of a multiplicative over an additive gradient descent is that the factors automatically retain the sign. One of the disadvantages is that elements of W or H, once zero, cannot become positive.

A.2 Technical Details

A.2.1 Software architectural details

Our software system comprises several key components, including:

- Database: for storing metadata and raw data
- Back-end: for processing requests and managing the framework
- Object storage: for storing raw data
- Worker: containerized analysis tools
- Graphical user interface: front-end

A schematic representation of these components is illustrated in Figure 3.8. The individual components can be deployed using a Docker Swarm cluster, as detailed in the appendix. The deployment process requires the execution of six terminal commands and the adjustment of certain configuration files to point to the address where the service is running.

In software architecture, the design and implementation of the high-level structure of the software is considered. The architecture is the result of combining specific architectural elements in specific forms to meet the functional and performance requirements, as well as other non-functional requirements such as reliability, scalability, portability, and availability. Kruchten's four plus one views is an approach for creating different views of a software system for different stakeholders. These views include:

- The logical view, which is the object model of the design (when an object-oriented design method is used)
- The process view, which captures the concurrency and synchronization aspects of the design
- The physical view, which describes the mapping of the software onto the hardware and reflects its distributed aspect
- The development view, which describes the static organization of the software in its development environment

To outline our system in Krutchen's views, we use the following views: logical view, process view, physical view, and development view. It should be noted that these views overlap in some of their responsibilities and some views may be omitted in certain cases. Overall, Software architecture can be represented as the combination of elements, forms, and rationale/constraints as stated by Perry and Wolfe in the formula [95]:

$$\text{Software architecture} = \text{Elements, Forms, Rationale/Constraints.}$$

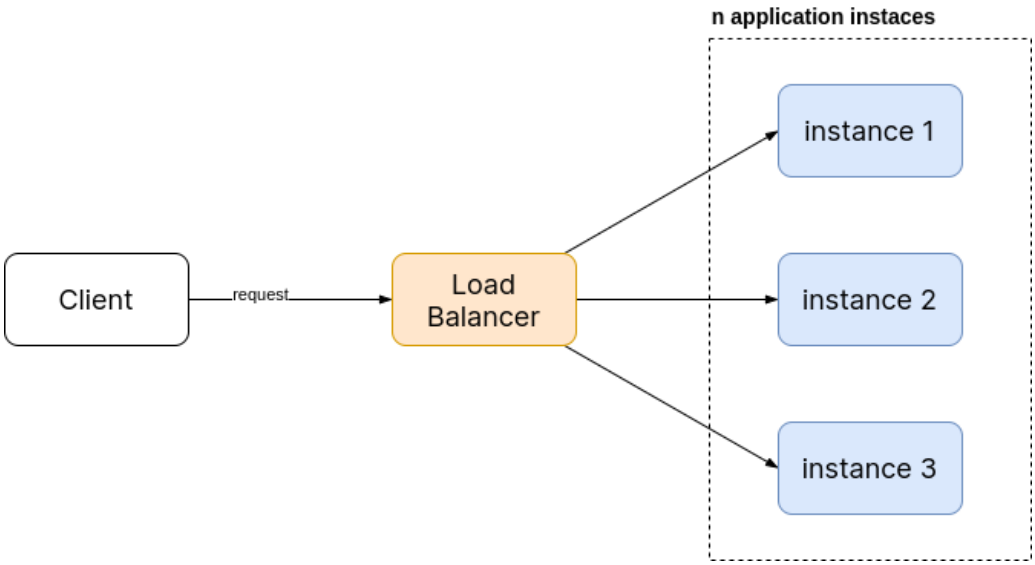


Figure A.1: This figure shows a load balancer using the Proxy design pattern. Incoming traffic is directed to the load balancer, which acts as an intermediary between clients and servers. The load balancer uses various algorithms to distribute the traffic to the available instances, ensuring that no single server becomes overloaded and improving overall system performance and availability.

Software patterns

Technical Setup Steps

In this section, we outline the process for setting up a system to integrate a custom script or third-party tool into a data analysis process. The following assumptions are made:

1. A set of computers that are within a secure network
2. A specific set of ports are open, as detailed in [7]

The following steps detail the setup procedure:

1. Install Docker on each machine in the cluster by following the official documentation [3]. On a Debian Linux system, the command is:

   ```
   apt-get install docker-ce docker-ce-cli containerd.io
   ```

2. Set up a Docker Swarm cluster by creating a manager with the command:

   ```
   docker swarm init --advertise-addr <MANAGER-IP>
   ```

   Then, add workers by running:

   ```
   docker swarm join --token <token> <MANAGER-IP>
   ```

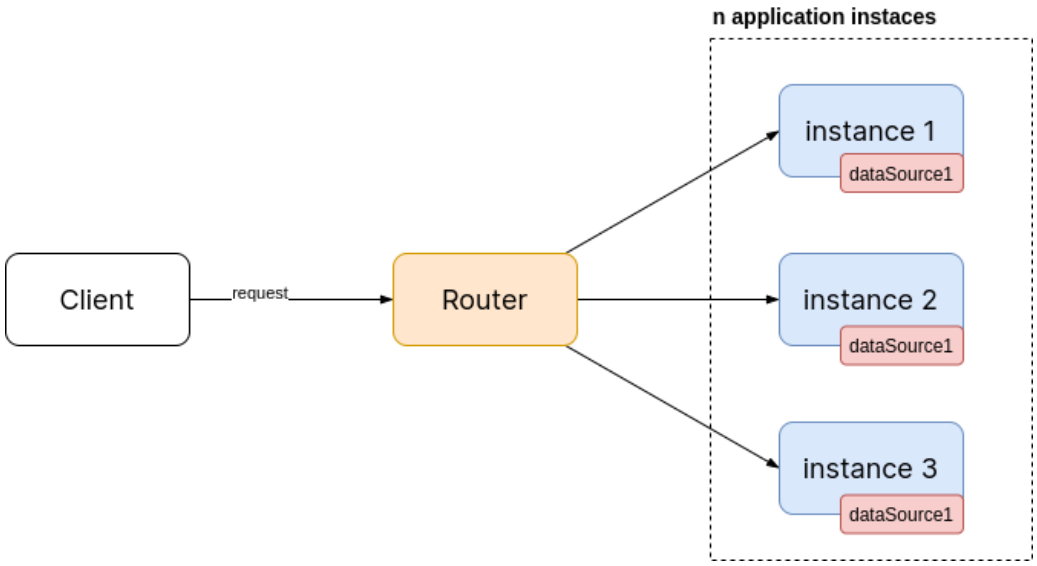


Figure A.2: Illustration of the Data-Driven Routing software design pattern in a router system. The diagram depicts the routing mechanism where the incoming requests are directed to the router module, which employs a data-driven approach to identify the appropriate instance and data source to handle the request. This pattern enables dynamic routing based on the characteristics of the request, thereby optimizing the utilization of resources and enhancing the overall system performance.

For further information, refer to the documentation [7].

3. Build a Docker container with the desired tool inside a Java wrapper. The Java application wraps around the tool and handles communication to the back end. In addition, this wrapper aggregates metadata, downloads raw data, and regularly checks jobs. For further information on Docker images, refer to the documentation [6]. The container can be built by running:

   ```
   docker build -t <tagName> .
   ```

4. Push the image to a registry by running:

   ```
   docker push <tagName>
   ```

5. Follow the deployment steps outlined in section Service deployment.

Note: The above instructions are for general setup process, the specific instructions to run on your current environment may vary.

Service deployment

The following commands are used to deploy the main parts of our proposed system. They depend on the software setup from the previous section being completed correctly.

```
docker service create --name=mariadb --publish=3306:3306 --with-registry
    ↪ -auth <registry>:5000/biomed/mariadbdocker
```

Then the pipeline backend can be deployed with the command:

```
docker service create --name=dropwizardbackend --publish=8080:8080 --
    ↪ with-registry-auth <registry>:5000/biomed/dropwizardbackend
```

The object storage is then deployed by

```
docker service create --name=minio --publish=9000:9000 --with-registry-
    ↪ auth -e "MINIO_ACCESS_KEY=<AKey>" -e "MINIO_SECRET_KEY=<sKey>" --
    ↪ mount source=data,t arget=/data minio/minio server /data;
```

The docker worker can be deployed by using:

```
docker service create --name=hmmsearch --publish=8084:8080 --with-
    ↪ registry-auth <registry>:5000/biomed/hmmsearchdocker:fuvm
```

Optionally a graphical user interface can be deployed with the command.

```
docker service create --name=vueplain --publish=5050:8080 --with-
    ↪ registry-auth <registry>:5000/biomed/vuejs-client-plain:fuvm
```

A.2.2 Example Task Deployment for hidden Markov model

After deployment of the cluster and associated services, this section outlines the technical steps for submitting a task. Specifically, this technical example utilizes a method known as Hidden Markov Model Search, which will be further detailed in Section Hidden Markov model search on prototypic sequences representing repetitive DNA from different eukaryotic species. The Hidden Markov Model search can be initiated through the use of both a RESTful API and a graphical user interface (GUI). Additionally, a simple web-based user interface (WebUI) is provided to facilitate task creation, wherein the user can upload data and select the desired algorithm for analysis. The system subsequently updates the task's status and notifies the user when results are available. Finally, the user can access and review the results. In addition to the GUI, this section also expands upon the use of terminal commands, as the graphical interface serves as a wrapper for the RESTful API.

1. In this example, two files must be uploaded to the system: a so-called profile file and a database file. For a file upload, the specific HTML request needs to include the Content-Disposition in the header to indicate the attachment. The request's response is a unique id of the uploaded file, which can be referenced in the following step. Using `curl` the request can be made as follows.

```
curl -X POST -F file=\@aRNH_profile.hmm <backendAddress>/
    ↪ upload
```

2. To create a job with the REST API, the request must include a JSON body, including the required parameters. In case of HMM search the `hmmprofile` and `fastadatabase` have to be included. Similar to the curl request in listing A.1. The response is a uniquely assigned jobId.

Listing A.1: HMM job curl request JSON body

```
curl --header "Content-Type: application/json" \
  --request POST \
  --data '\
{\
  "algorithmId": "hmmsearch",\
  "parameter": [\
    {\
      "name": "hmmprofile",\
      "content": "fil_ff44c192a5"\
    },\
    {\
      "name": "fastadatabase",\
      "content": "fil_027784d4a0"\
    }\
  ],\
  "bucket": "newbucket",\
  "path": ""\
}\
  <backendAdress>/job
```

3. Finally, the job status can be checked with the unique jobId. By using curl this can be queried as follows

```
curl <backendAdress>/job/<jobId>
```

Workflow Code

Listing A.2: HMM Workflow Definition

```
const profiles : string[]= await this.storage.glob( args["hmm"] )
  .then( profiles => {
    return profiles.map( meta => {
    return meta.id;
  });
});

const genes :string[] = await this.storage.glob( args["fasta"] )
  .then( fasta => {
    return fasta.map( meta => {
    return meta.id;
  });
});

```

```
    const hmm = await this.pipeline.init(genes)
      .apply('dnatoprotein', key => {
        return [{ name: 'dnasequence',
          content: (key as string[])[0]
        }];
      })
      .then( ( next : NyetusPipeline ) => {
        return next.flatten( ( key : [{ [key: string]: string }] )=> {
          const cross : Result[] = profiles.map( profile => {
          const join : Result = [{}];
              join[0]["fastadatabase"] = key[0]['file_id'];
              join[0]["hmmprofile"] = profile;
          return join;
    label={lst:Dockerfile}]

    FROM openjdk:8-jre
RUN apt-get update -y && apt-get install -y \
    hmmer
COPY config.yml /opt/hmmerSearch/
COPY build/libs/HmmDocker.jar /opt/hmmerSearch/
EXPOSE 8080
WORKDIR /opt/hmmerSearch
 CMD ["java", "-Xms128m", "-Xmx1500m", "-Dfile.encoding=UTF-8",
 "-jar", "HmmDocker.jar", "server", "config.yml"]
```

Listing A.3: REST json body

```
"piplineName": "hmmserach ",
"tasks": [{
        "taskname": "DNAtranslate",
        "dependentTask": "",
        "output":"translatedDna.fasta",
        "task": [{
            "algorithmId": "dnatoprotein",
            "parameter": [{
                    "name": "dnasequence",
                    "content": "fil_ff4a5"
                }
            ],
            "bucket": "newbucket",
            "path": "path/to/"
        }]
    },
    {
        "taskname": "Hmmsearch",
        "dependentTask": "DNAtranslate",
        "output":"hmmsearchresults.txt",
        "task": [{
            "algorithmId": "hmmsearch",
            "parameter": [{
                    "name": "hmmprofile",
                    "content": "fil_ff41c2a5"
```

```
26                 },
27                 {
28                     "name": "fastadatabase",
29                     "content": "fil_02778bba0"
30                 }
31             ],
32             "bucket": "newbucket",
33             "path": "path/to/"
34         }]
35     }
36 ]
```

A.2.3 Linux containerisation

Name	Description
Docker	is an open-source management tool for containers that automates the deployment of applications. Docker uses a client-server architecture consisting of three main components: Docker client, Docker host and Docker registry. Docker host represents the hosting machine on which Docker daemon and containers run. The Docker daemon is responsible for building, running, and distributing the containers. The Docker client is the user interface to Docker.
Rocket (rkt)	is an emerging new container technology. With the advent of CoreOS [189], a new container called Rocket was introduced. Besides rkt containers, CoreOS supports Docker. Rocket was designed to be a more secure, interoperable, and open container solution. Rocket is a new competitor for Docker.
Linux Containers [190]	is an operating system-level virtualization method for running multiple isolated Linux systems. It uses kernel-level namespaces to isolate the container from the host.
Linux Container Daemon [191]	is a lightweight hypervisor designed by Canonical, for Linux containers are built on top of LXC to provide a new and better user experience. LXD and Docker make use of LXC containers.

Table A.1: This table overviews the critical aspects of popular Linux virtualization technologies, including Docker, Rocket, Linux Containers (LXC), and Linux Container Daemon (LXD). It summarizes the features and benefits of each technology and highlights how they can be utilized in different scenarios.

A.3 Additions to experiments

A.3.1 Matrix decomposition addtions

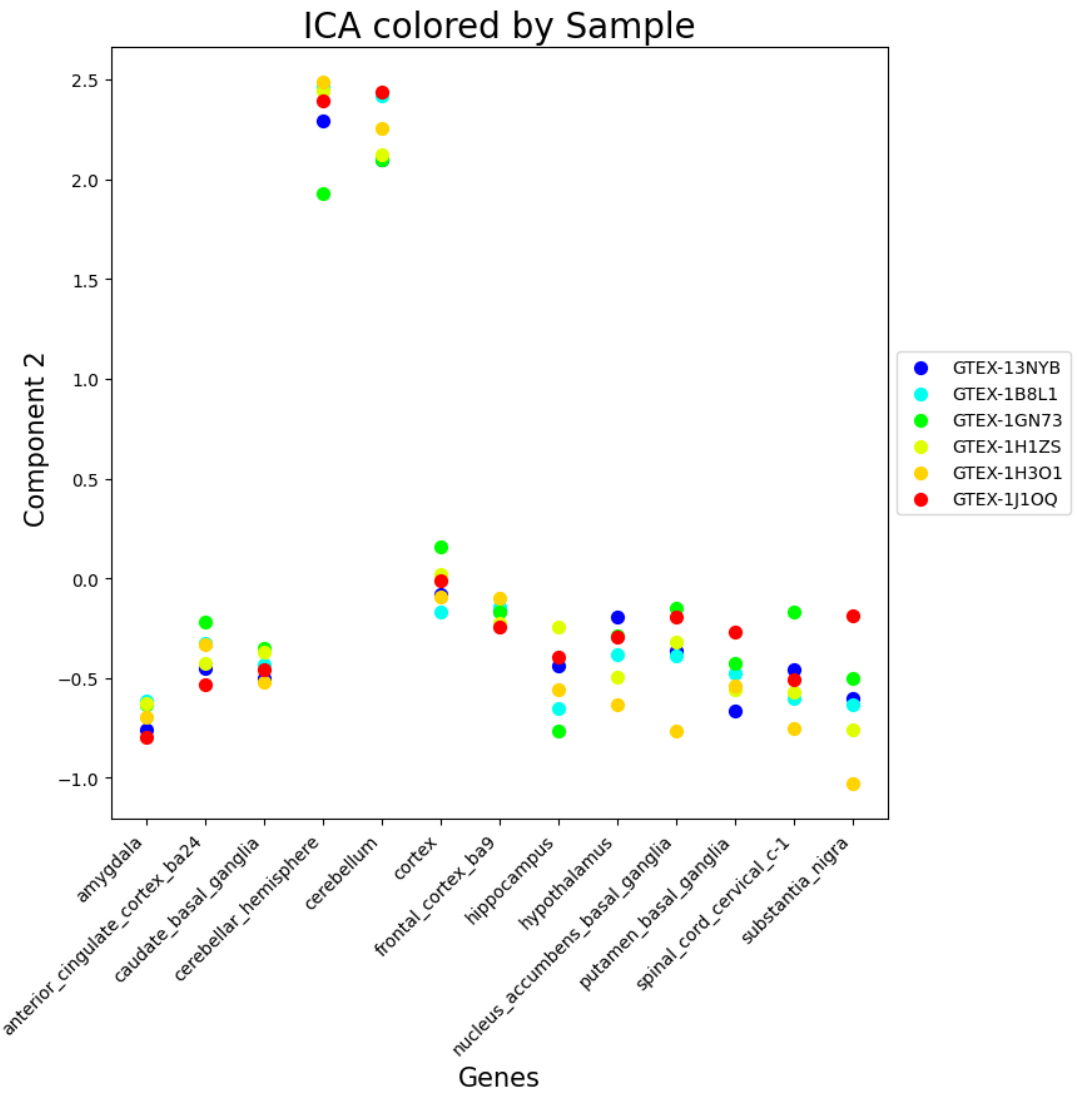


Figure A.3: independent component analysis

A.3.2 Hidden Markov model additions

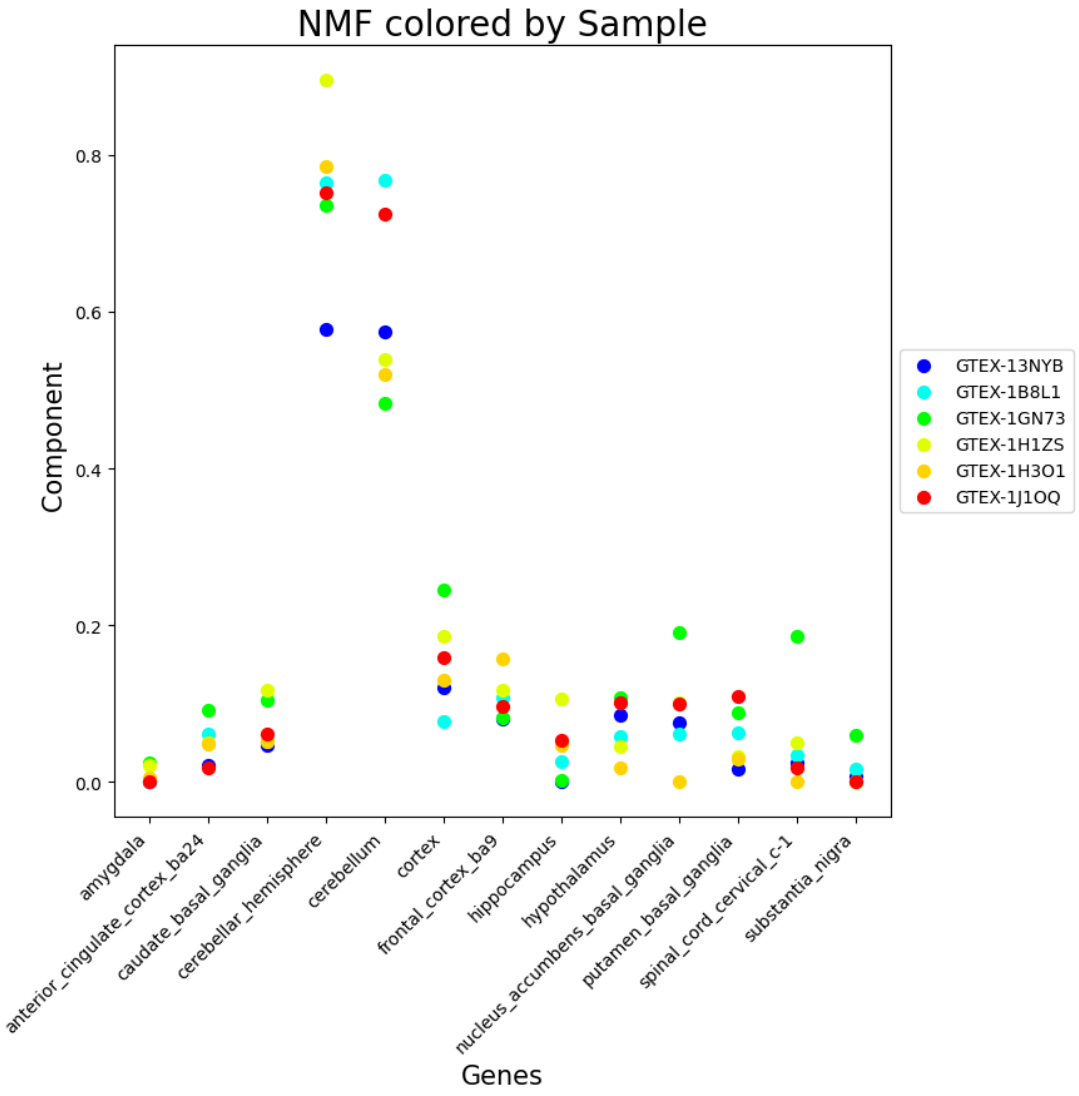
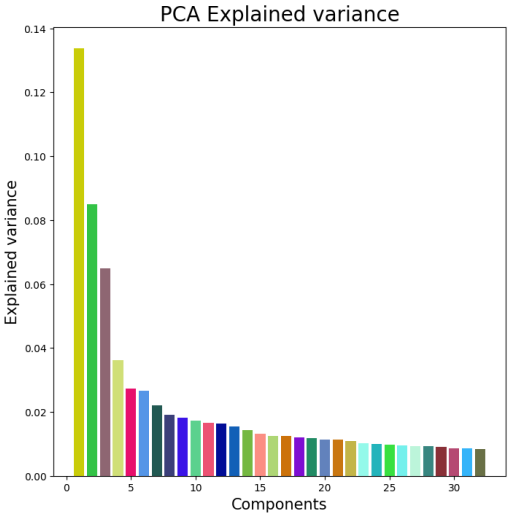
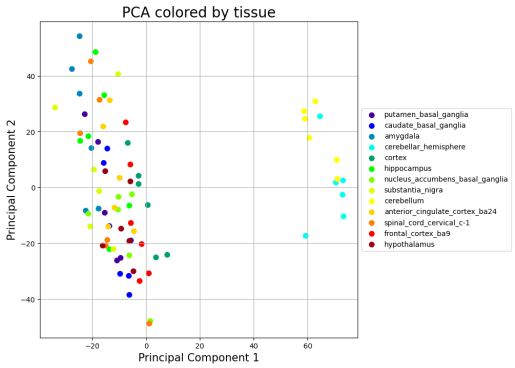


Figure A.4: non-negative matrix factorisation



(a) Explained variance



(b) The figure presents a scatter plot generated using principal component analysis. In this plot, the x-axis and y-axis represent the two principal components of the data. These components are the resulting dimensions from the PCA analysis, and the scatter plot allows for the visualization of the relationship between the different variables in the data. Each data point in the plot corresponds to a single observation, and the position of the point reflects the values of the variables for that observation on the x-axis and y-axis.

Figure A.5: The figure illustrates the results of applying principal component analysis (PCA) to the data. The figure (a)illustrates the explained variance of the data when using principal component analysis (PCA). Explained variance refers to the amount of variance in the data that is accounted for by each principal component. The figure (b) depicts a scatter plot generated using PCA. A scatter plot is a graphical representation of two or more variables, where each individual data point is plotted as a point in the graph. In this case, the scatter plot has been generated using PCA to visualize the relationship between the different variables in the data.

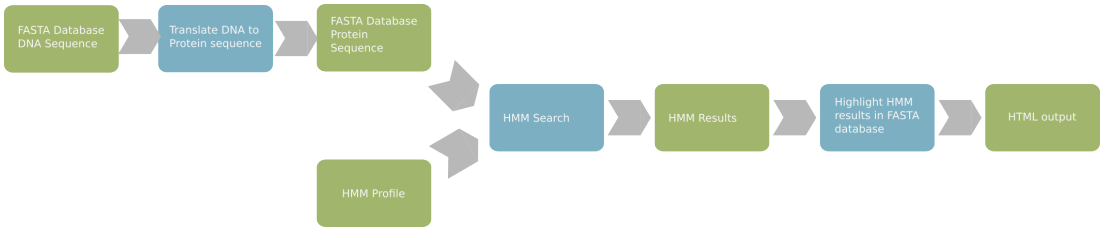


Figure A.6: hidden Markov model pipeline for protein sequence analysis. The process begins with a fasta database containing multiple protein sequences. The sequences are then analyzed using a hidden Markov model (HMM) to identify conserved domains and potential functional sites. The results are then visualized through a variety of tools, such as sequence logos and domain architecture diagrams, to provide insight into the structural and functional characteristics of the proteins in the database.

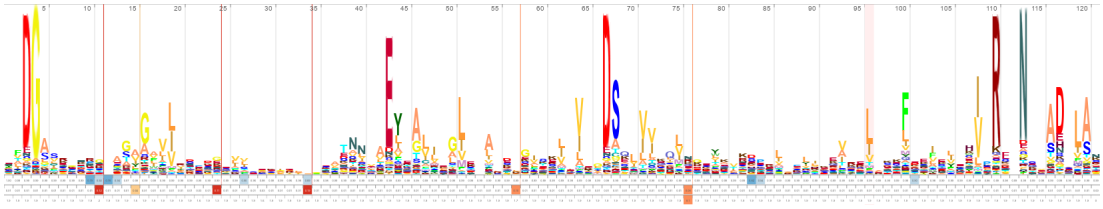


Figure A.7: aRNH sequence logo section represents the relative frequency of nucleotides at each position in the hairpin loop, with the height of each letter indicating the degree of conservation at that position. The overall shape of the logo provides insight into the structural characteristics of the hairpin loop, including the location and strength of base-pairing interactions.

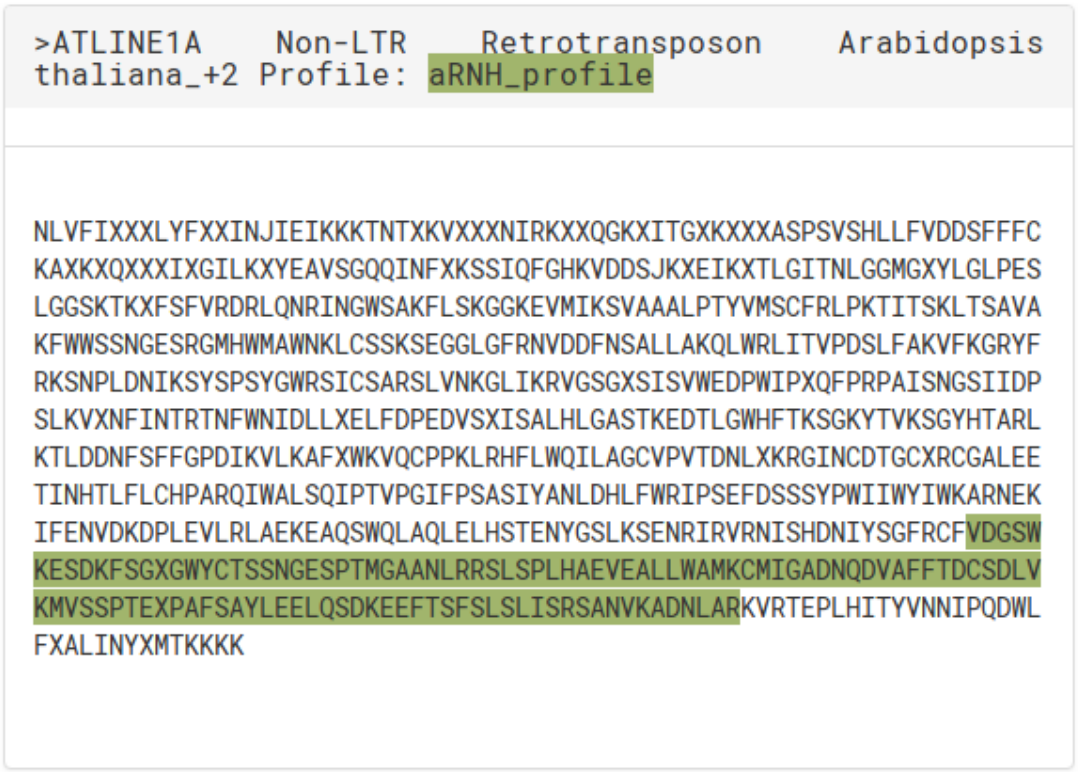


Figure A.8: Visualization of HMM search results displaying the protein sequence with the corresponding HMM profile overlaid. The HMM profile is represented by the colored bars above the sequence, with each color corresponding to a different state in the HMM model. The sequence positions that match with the HMM profile are highlighted in green, emphasizing the correspondence between the protein sequence and the HMM profile.